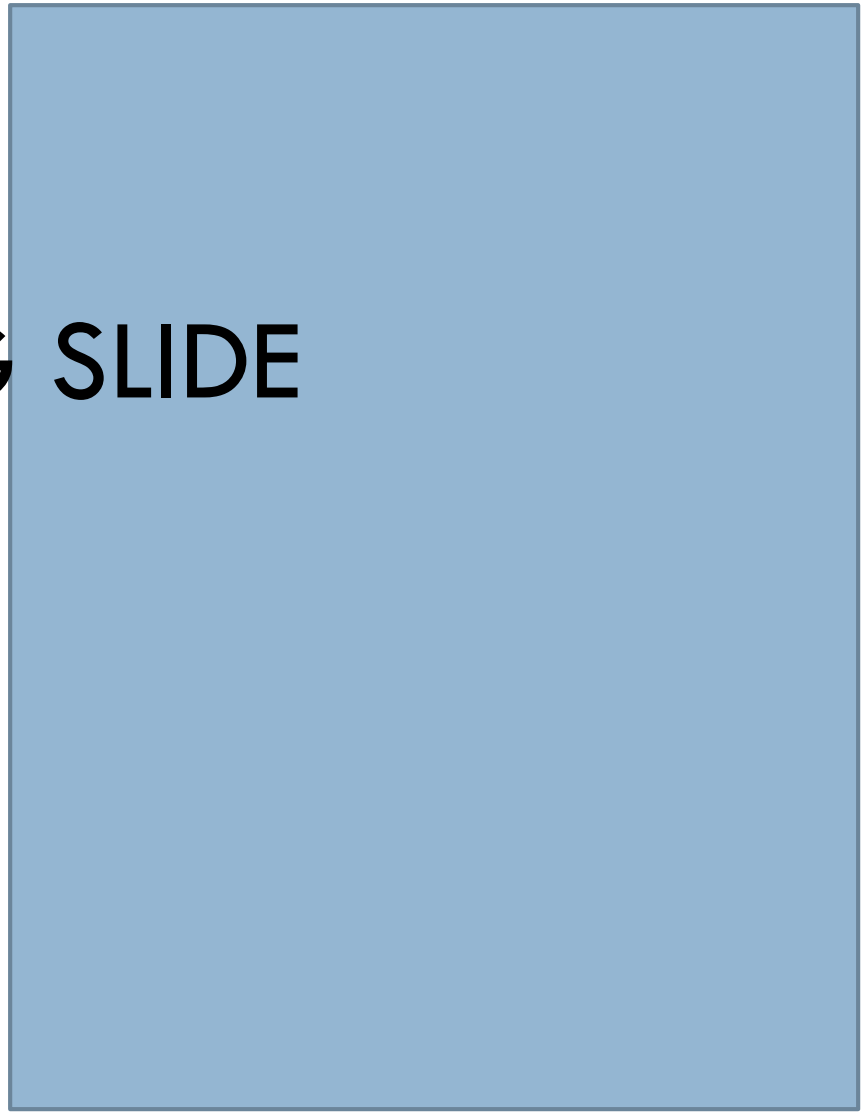
TUNING SLIDE



```
Cycles:      1.4 B    Sim Speed: 172.4 MCPS    Avg Act Cores: 1.00
Instrs:      1.3 B    Sim Speed: 169.2 MIPS    Avg Core IPC:  0.98
```

Fast and Accurate Microarchitectural Simulation with ZSim

Daniel Sanchez, Nathan Beckmann,
Anurag Mukkara, Po-An Tsai

MIT CSAIL

MICRO-48 Tutorial
December 5, 2015

Massachusetts Institute of Technology

Agenda

8:30 – 9:10	Intro and Overview
9:10 – 9:25	Simulator Organization
9:25 – 10:00	Core Models
10:00 – 10:20	Break / Q&A
10:20 – 11:00	Memory System
11:00 – 11:20	Configuration and Stats
11:20 – 11:40	Validation
11:40 – 12:00	Q&A

Introduction and Overview

Massachusetts Institute of Technology

Motivation

- Current detailed simulators are slow (~200 KIPS)

Motivation

- Current detailed simulators are slow (~200 KIPS)
- Simulation performance wall
  - More complex targets (multicore, memory hierarchy, …)
  - Hard to parallelize

Motivation

- Current detailed simulators are slow (~200 KIPS)
- Simulation performance wall
  - More complex targets (multicore, memory hierarchy, …)
  - Hard to parallelize
- Problem: Time to simulate 1000 cores @ 2GHz for 1s at
  - 200 KIPS: **4 months**

Motivation

- Current detailed simulators are slow (~200 KIPS)
- Simulation performance wall
  - More complex targets (multicore, memory hierarchy, …)
  - Hard to parallelize
- Problem: Time to simulate 1000 cores @ 2GHz for 1s at
  - 200 KIPS: **4 months**
  - 200 MIPS: **3 hours**

Motivation

- Current detailed simulators are slow (~200 KIPS)
- Simulation performance wall
  - More complex targets (multicore, memory hierarchy, …)
  - Hard to parallelize
- Problem: Time to simulate 1000 cores @ 2GHz for 1s at
  - 200 KIPS: **4 months**
  - 200 MIPS: **3 hours**
- Alternatives?
  - FPGAs: Fast, good progress, but still hard to use
  - Simplified/abstract models: Fast but inaccurate

ZSim Techniques

- Three techniques to make 1000-core simulation practical:
  1. Detailed DBT-accelerated core models to speed up sequential simulation
  2. Bound-weave to scale parallel simulation
  3. Lightweight user-level virtualization to bridge user-level/full-system gap

ZSim Techniques

- Three techniques to make 1000-core simulation practical:
  1. Detailed DBT-accelerated core models to speed up sequential simulation
  2. Bound-weave to scale parallel simulation
  3. Lightweight user-level virtualization to bridge user-level/full-system gap

- ZSim achieves high performance and accuracy:
  - Simulates 1024-core systems at 10s-1000s of MIPS
  - 100-1000x faster than current simulators
  - Validated against real Westmere system, avg error ~10%

This Presentation is Also a Demo!

- ZSim is simulating these slides
  - OOO Westmere cores running at 2 GHz
  - 3-level cache hierarchy
- Will illustrate other features as I present them

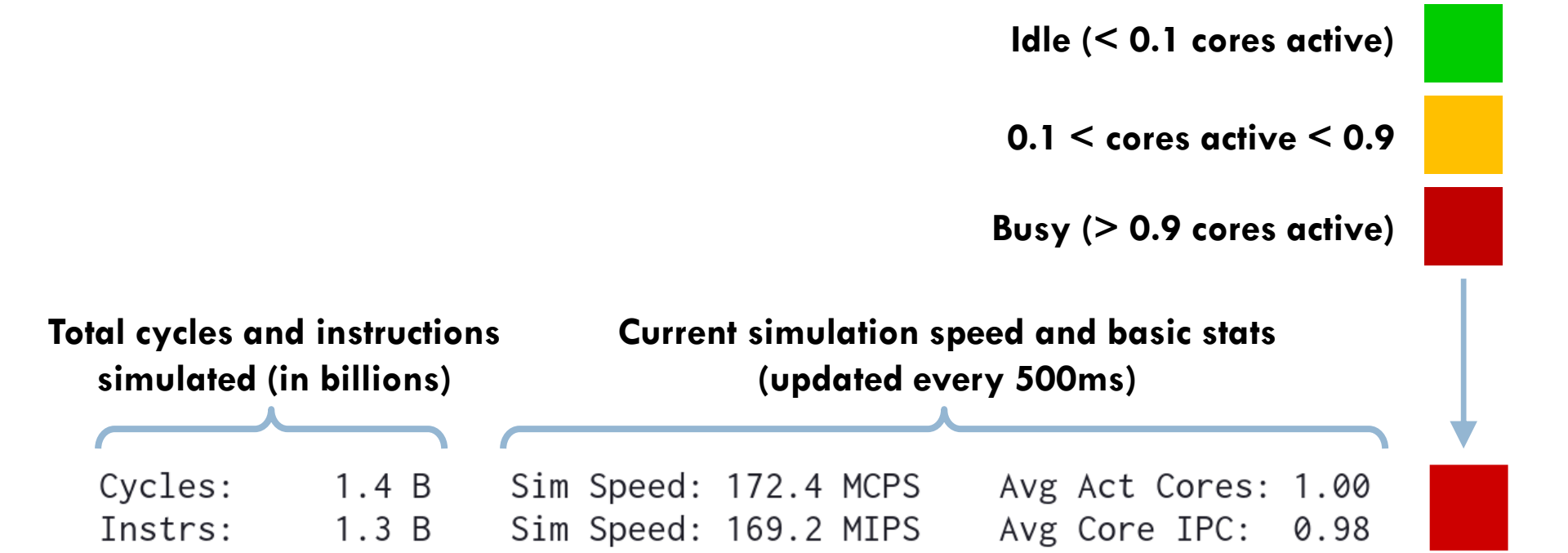
Total cycles and instructions simulated (in billions)

Current simulation speed and basic stats (updated every 500ms)

```
Cycles:     1.4 B    Sim Speed: 172.4 MCPS    Avg Act Cores: 1.00
Instrs:     1.3 B    Sim Speed: 169.2 MIPS    Avg Core IPC:  0.98
```

This Presentation is Also a Demo!

- ZSim is simulating these slides
  - OOO Westmere cores running at 2 GHz
  - 3-level cache hierarchy
- Will illustrate other features as I present them



This Presentation is Also a Demo!

- ZSim is simulating these slides
  - OOO Westmere cores running at 2 GHz
  - 3-level cache hierarchy
- Will illustrate other features as I present them

ZSim performance relevant when busy
Running on 2-core laptop CPU @ 1.7 GHz
~12x slower than 16-core server @ 2.6 GHz

Idle (< 0.1 cores active)

0.1 < cores active < 0.9

Busy (> 0.9 cores active)

Total cycles and instructions simulated (in billions)

Current simulation speed and basic stats (updated every 500ms)

```
Cycles:      1.4 B    Sim Speed: 172.4 MCPS    Avg Act Cores: 1.00
Instrs:      1.3 B    Sim Speed: 169.2 MIPS    Avg Core IPC:  0.98
```

Main Design Decisions

- General execution-driven simulator:

Main Design Decisions

- General execution-driven simulator:

Emulation? (e.g., gem5, MARSSx86)
Instrumentation? (e.g., Graphite, Sniper)

Main Design Decisions

- General execution-driven simulator:

Emulation? (e.g., gem5, MARSSx86)

Instrumentation? (e.g., Graphite, Sniper)

Dynamic Binary Translation (Pin)

✓ Functional model "for free"

✗ Base ISA = Host ISA (x86)

Main Design Decisions

- General execution-driven simulator:

Emulation? (e.g., gem5, MARSSx86)
Instrumentation? (e.g., Graphite, Sniper)

Cycle-driven?
Event-driven?

Dynamic Binary Translation (Pin)

✓ Functional model "for free"

✗ Base ISA = Host ISA (x86)

Main Design Decisions

- General execution-driven simulator:

Emulation? (e.g., gem5, MARSSx86)
Instrumentation? (e.g., Graphite, Sniper)

Cycle-driven?
Event-driven?

Dynamic Binary Translation (Pin)

✓ Functional model "for free"

✗ Base ISA = Host ISA (x86)

DBT-accelerated, instruction-driven core

+

Event-driven uncore

Outline

- Introduction
- Detailed DBT-accelerated core models
- Bound-weave parallelization
- Lightweight user-level virtualization

Accelerating Core Models

- Shift most of the work to DBT instrumentation phase

Basic block

```
mov   (%rbp),%rcx
add   %rax,%rbx
mov   %rdx,(%rbp)
ja   40530a
```

Instrumented basic block + **Basic block descriptor**

```
Load(addr = (%rbp))
mov   (%rbp),%rcx
add   %rax,%rdx
Store(addr = (%rbp))
mov   %rdx,(%rbp)
BasicBlock(BBLDescriptor)
ja   10840530a
```

Ins→µop decoding
µop dependencies,
functional units, latency
Front-end delays

Accelerating Core Models

- Shift most of the work to DBT instrumentation phase

Basic block

```
mov   (%rbp),%rcx
add   %rax,%rbx
mov   %rdx,(%rbp)
ja   40530a
```

Instrumented basic block + **Basic block descriptor**

```
Load(addr = (%rbp))
mov   (%rbp),%rcx
add   %rax,%rdx
Store(addr = (%rbp))
mov   %rdx,(%rbp)
BasicBlock(BBLDescriptor)
ja   10840530a
```

Ins→µop decoding
µop dependencies, functional units, latency
Front-end delays

- Instruction-driven models: Simulate all stages at once for each instruction/ µop

Accelerating Core Models

- Shift most of the work to DBT instrumentation phase

Basic block

```
mov   (%rbp),%rcx
add   %rax,%rbx
mov   %rdx,(%rbp)
ja   40530a
```

Instrumented basic block + **Basic block descriptor**

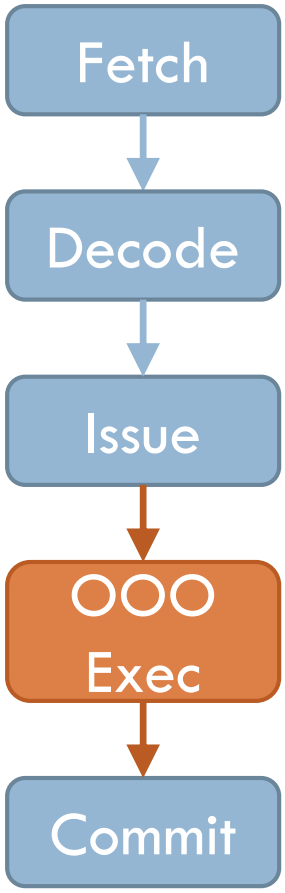
```
Load(addr = (%rbp))
mov   (%rbp),%rcx
add   %rax,%rdx
Store(addr = (%rbp))
mov   %rdx,(%rbp)
BasicBlock(BBLDescriptor)
ja   10840530a
```

Ins→μop decoding
μop dependencies, functional units, latency
Front-end delays

- Instruction-driven models: Simulate all stages at once for each instruction/ μop
  - Accurate even with OOO if instruction window prioritizes older instructions
  - Faster, but more complex than cycle-driven

Detailed OOO Model

- OOO core modeled and validated against Westmere



Main Features

Wrong-path fetches
Branch Prediction

Front-end delays (predecoder, decoder)
Detailed instruction to µop decoding

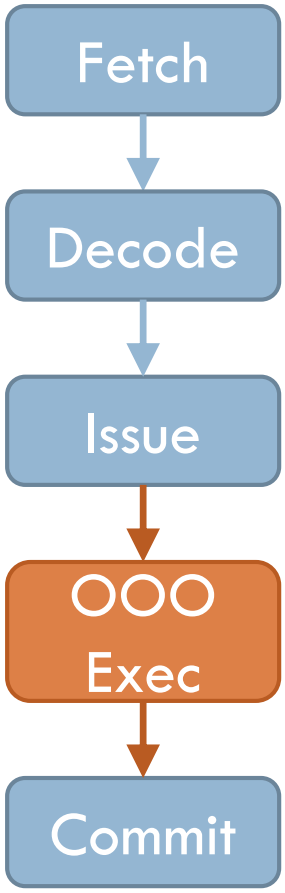
Rename/capture stalls
IW with limited size and width

Functional unit delays and contention
Detailed LSU (forwarding, fences,…)

Reorder buffer with limited size and width

Detailed OOO Model

- OOO core modeled and validated against Westmere

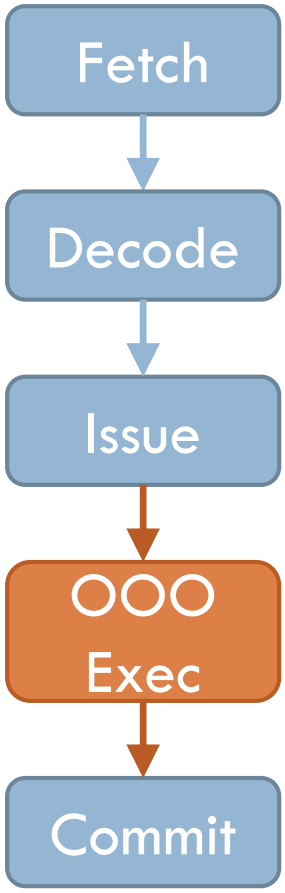


Detailed OOO Model

- OOO core modeled and validated against Westmere

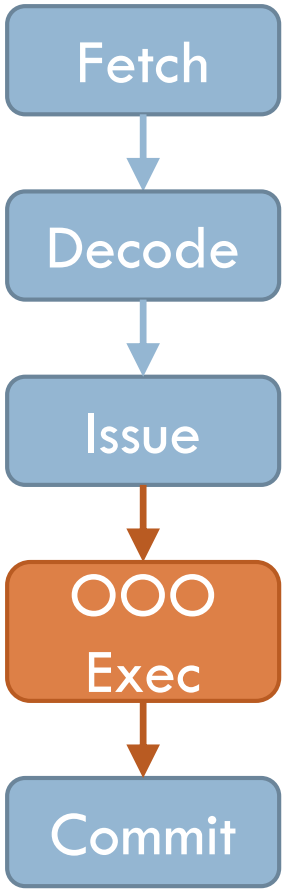
Fundamentally Hard to Model

Wrong-path execution



Detailed OOO Model

- OOO core modeled and validated against Westmere



Fundamentally Hard to Model

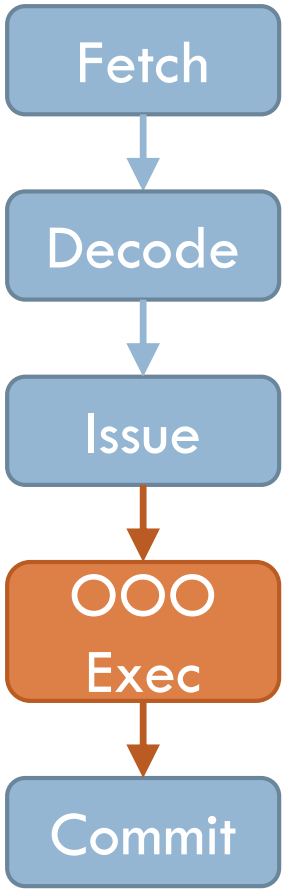
Wrong-path execution

In Westmere, wrong-path instructions don't affect recovery latency or pollute caches

Skipping OK

Detailed OOO Model

- OOO core modeled and validated against Westmere



Fundamentally Hard to Model

Wrong-path execution

In Westmere, wrong-path instructions don't affect recovery latency or pollute caches

Skipping OK

Not Modeled (Yet)

Rarely used instructions

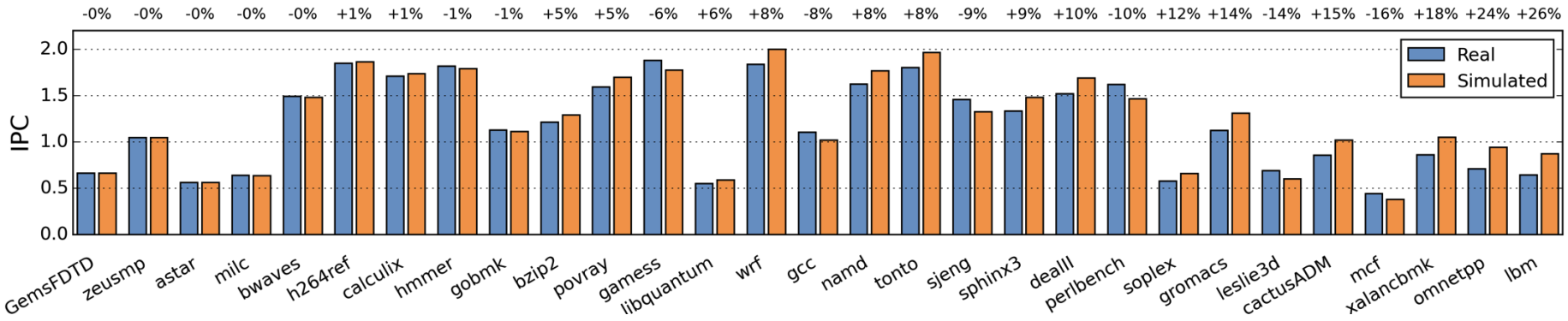
BTB

LSD

TLBs

Single-Thread Accuracy

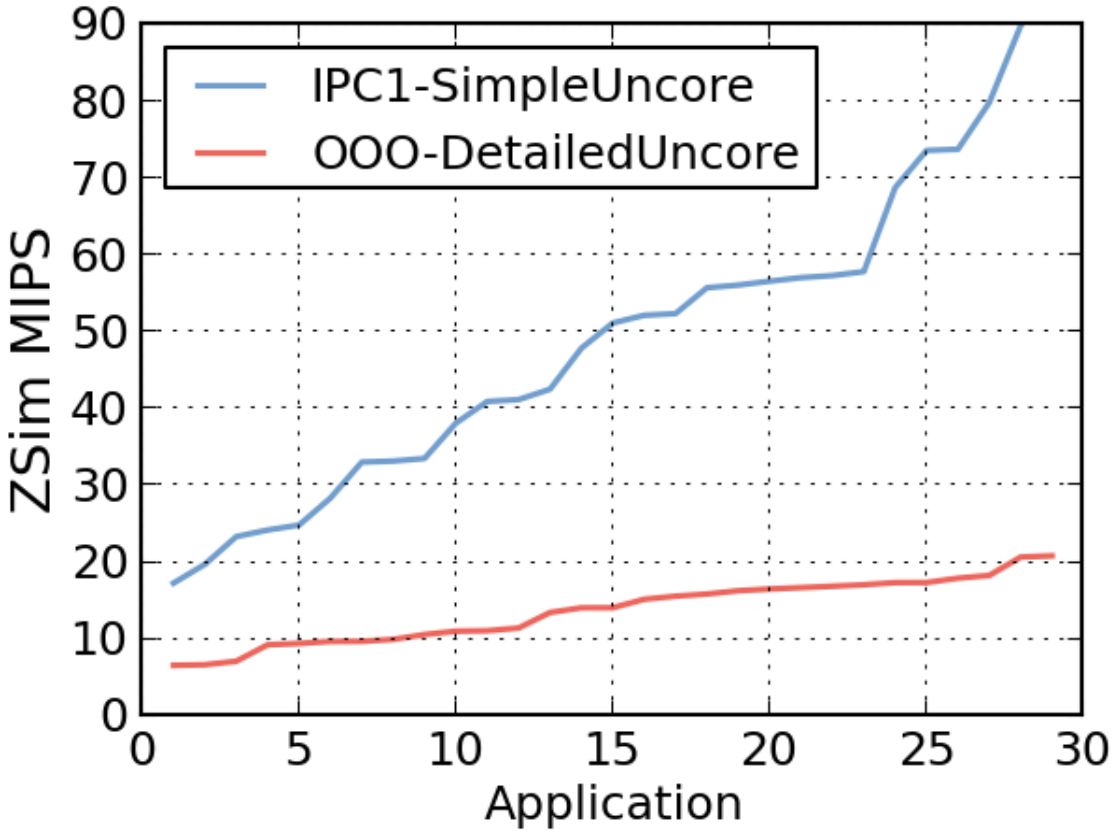
- 29 SPEC CPU2006 apps for 50 Billion instructions
- Real: Xeon L5640 (Westmere), 3x DDR3-1333, no HT
- Simulated: OOO cores @ 2.27 GHz, detailed uncore



- 8.5% average IPC error, max 26%, 21/29 within 10%

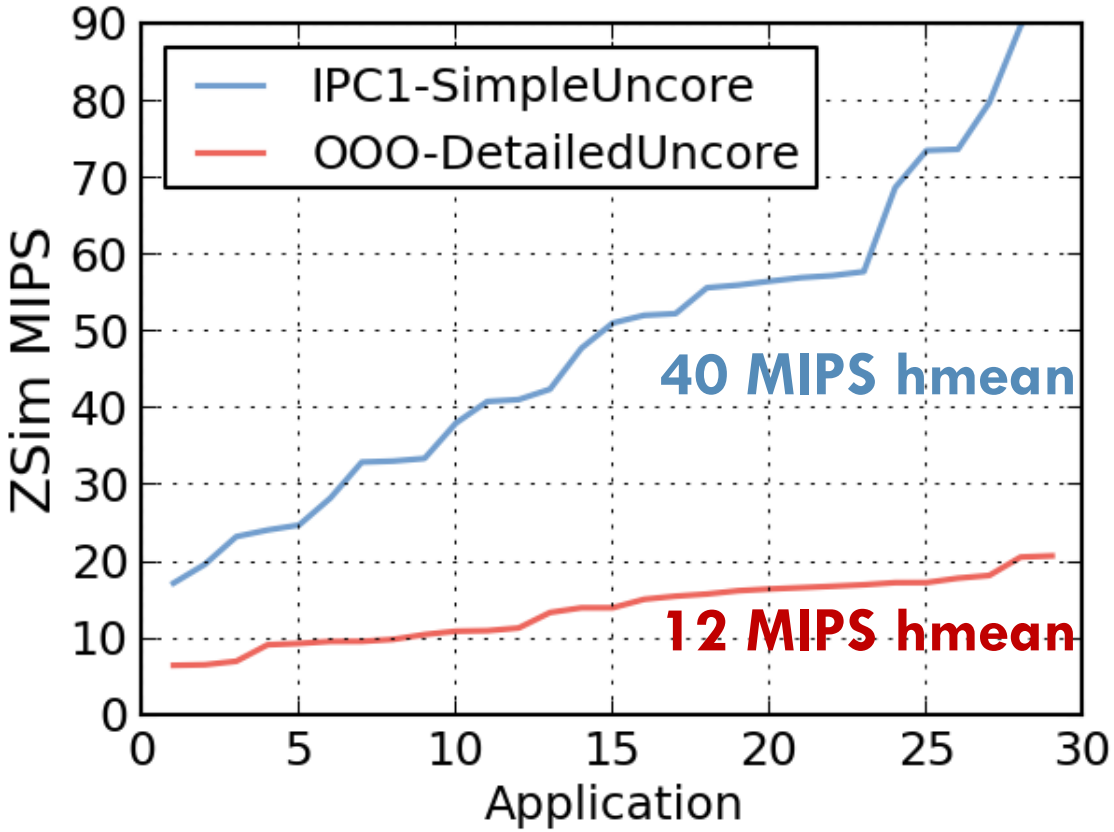
Single-Thread Performance

- Host: E5-2670 @ 2.6 GHz (single-thread simulation)
- 29 SPEC CPU2006 apps for 50 Billion instructions



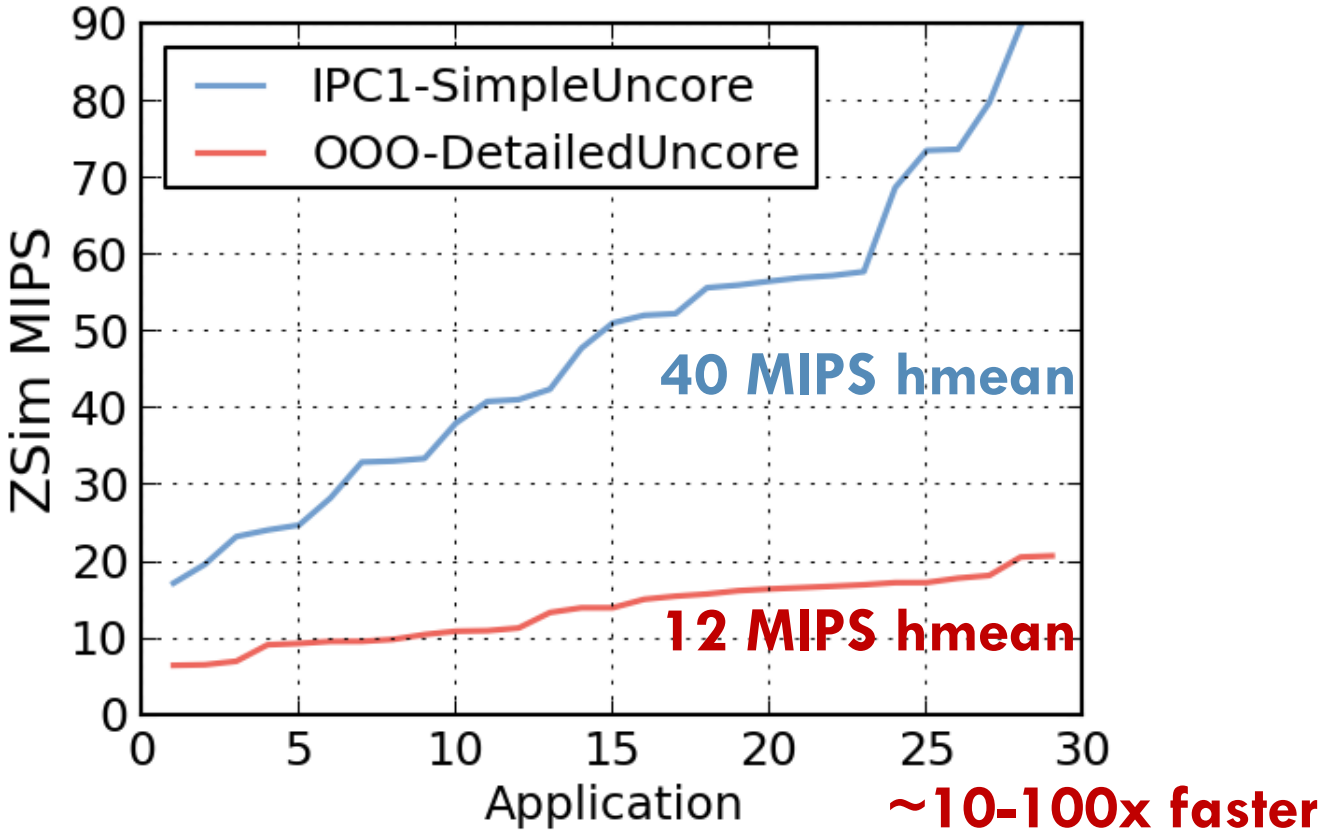
Single-Thread Performance

- Host: E5-2670 @ 2.6 GHz (single-thread simulation)
- 29 SPEC CPU2006 apps for 50 Billion instructions



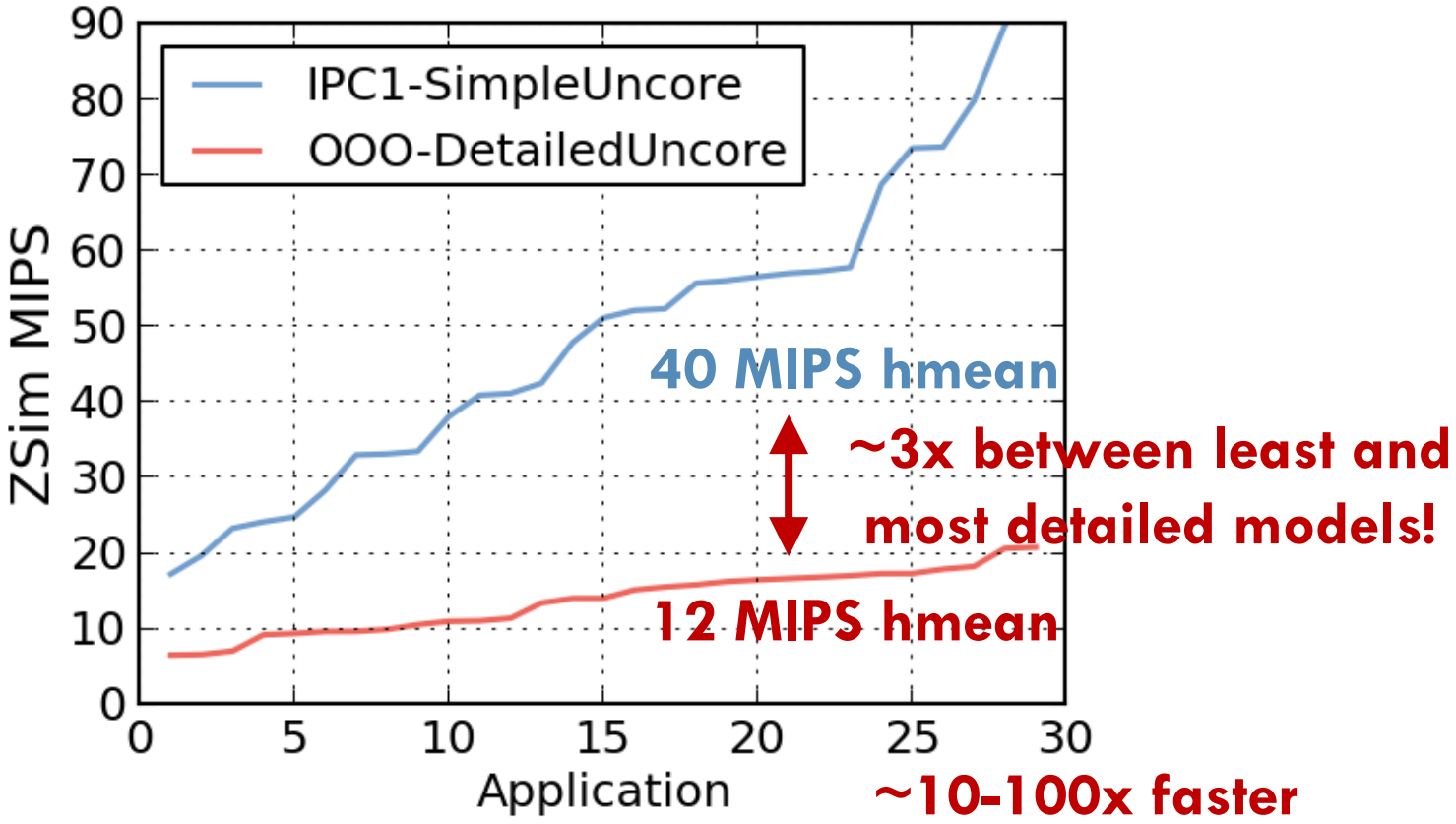
Single-Thread Performance

- Host: E5-2670 @ 2.6 GHz (single-thread simulation)
- 29 SPEC CPU2006 apps for 50 Billion instructions



Single-Thread Performance

- Host: E5-2670 @ 2.6 GHz (single-thread simulation)
- 29 SPEC CPU2006 apps for 50 Billion instructions

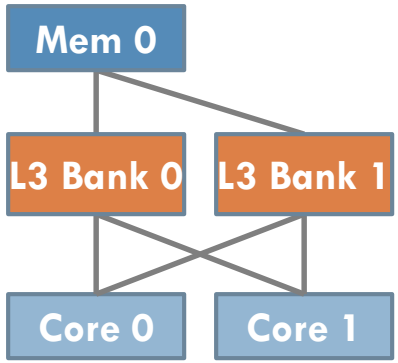


Outline

- Introduction
- Detailed DBT-accelerated core models
- Bound-weave parallelization
- Lightweight user-level virtualization

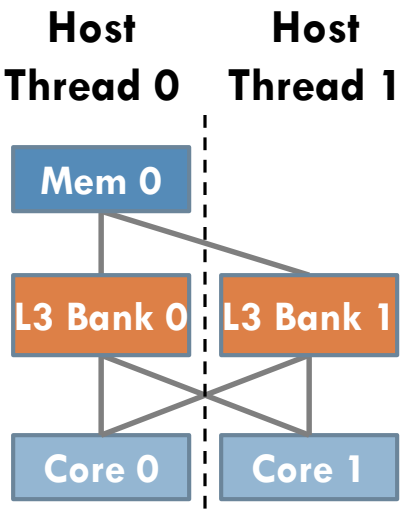
Parallelization Techniques

- Parallel Discrete Event Simulation (PDES):
  - Divide components across host threads
  - Execute events from each component maintaining illusion of full order



Parallelization Techniques

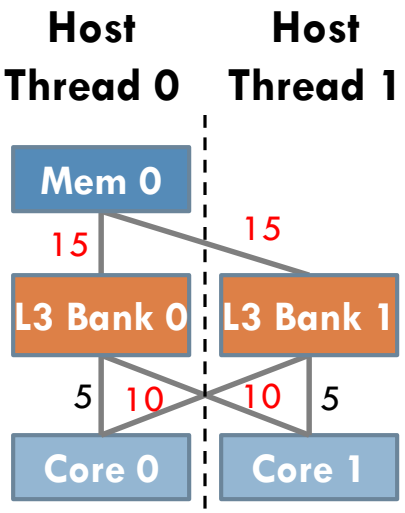
- Parallel Discrete Event Simulation (PDES):
  - Divide components across host threads
  - Execute events from each component maintaining illusion of full order



Parallelization Techniques

- Parallel Discrete Event Simulation (PDES):
  - Divide components across host threads
  - Execute events from each component maintaining illusion of full order

Skew < 10 cycles



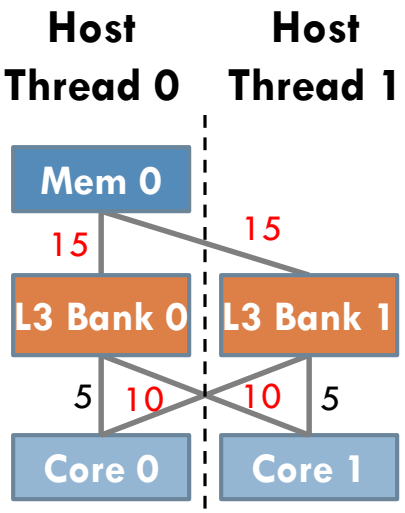
Parallelization Techniques

- Parallel Discrete Event Simulation (PDES):
  - Divide components across host threads
  - Execute events from each component maintaining illusion of full order

✓ Accurate

✘ Not scalable

Skew < 10 cycles



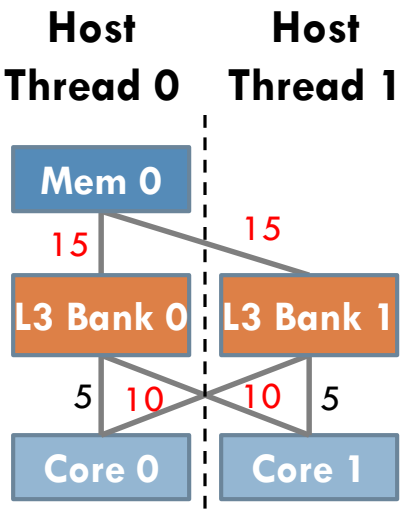
Parallelization Techniques

- Parallel Discrete Event Simulation (PDES):
  - Divide components across host threads
  - Execute events from each component maintaining illusion of full order

✓ Accurate

✘ Not scalable

Skew < 10 cycles



- Lax synchronization: Allow skews above inter-component latencies, tolerate ordering violations

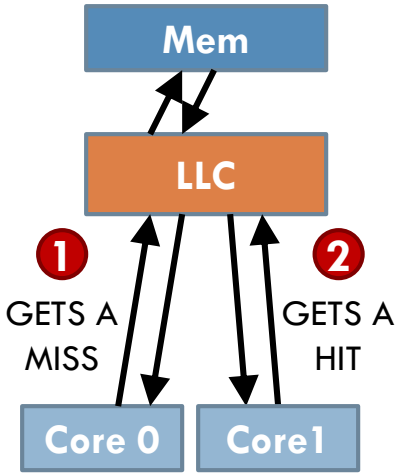
✓ Scalable

✘ Inaccurate

Characterizing Interference

Path-altering interference

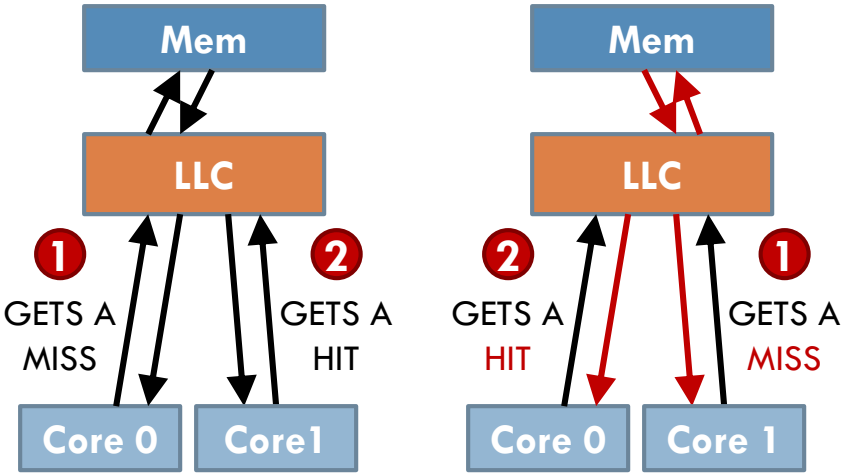
If we simulate two accesses out of order, their paths through the memory hierarchy change



Characterizing Interference

Path-altering interference

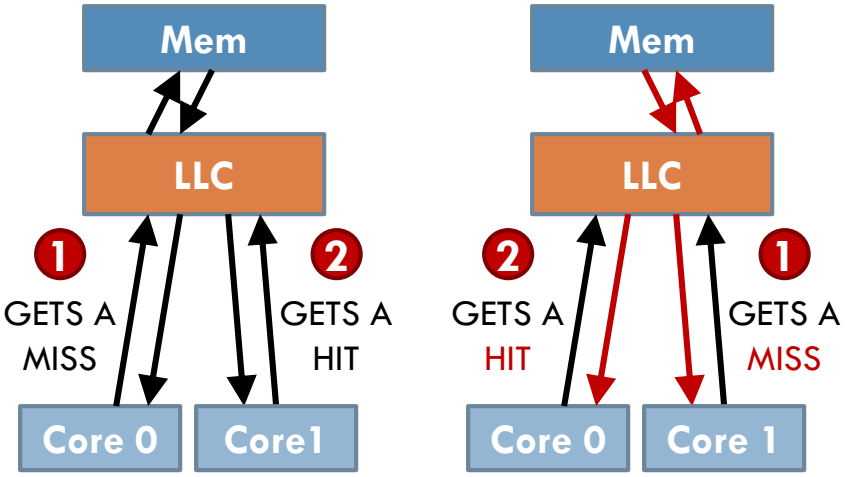
If we simulate two accesses out of order, their paths through the memory hierarchy change



Characterizing Interference

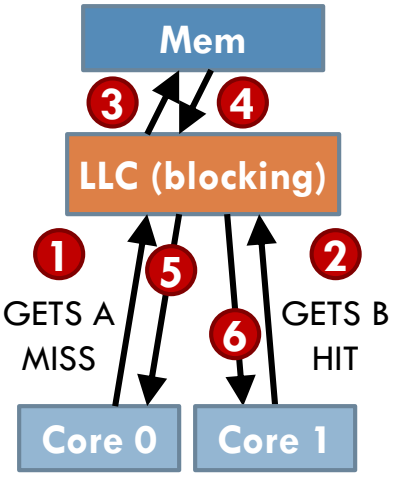
Path-altering interference

If we simulate two accesses out of order, their paths through the memory hierarchy change



Path-preserving interference

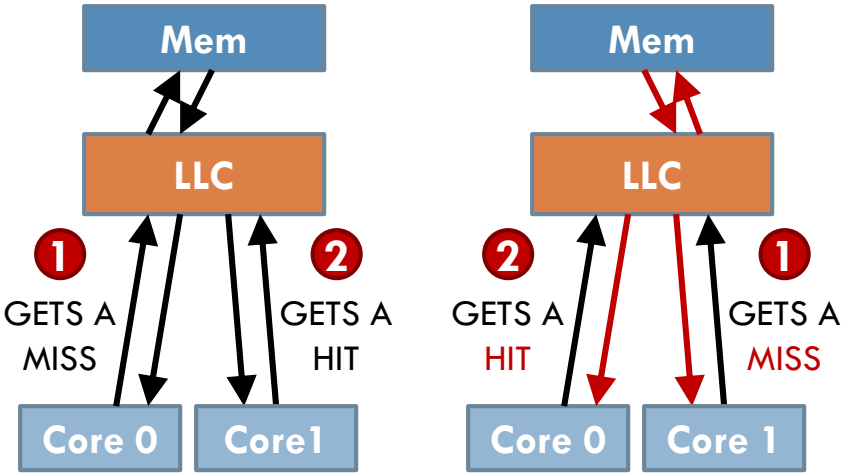
If we simulate two accesses out of order, their timing changes but their paths do not



Characterizing Interference

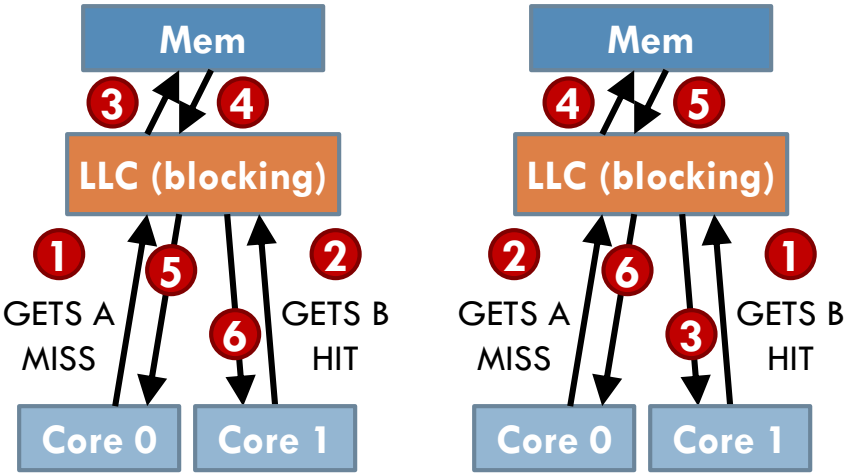
Path-altering interference

If we simulate two accesses out of order, their paths through the memory hierarchy change



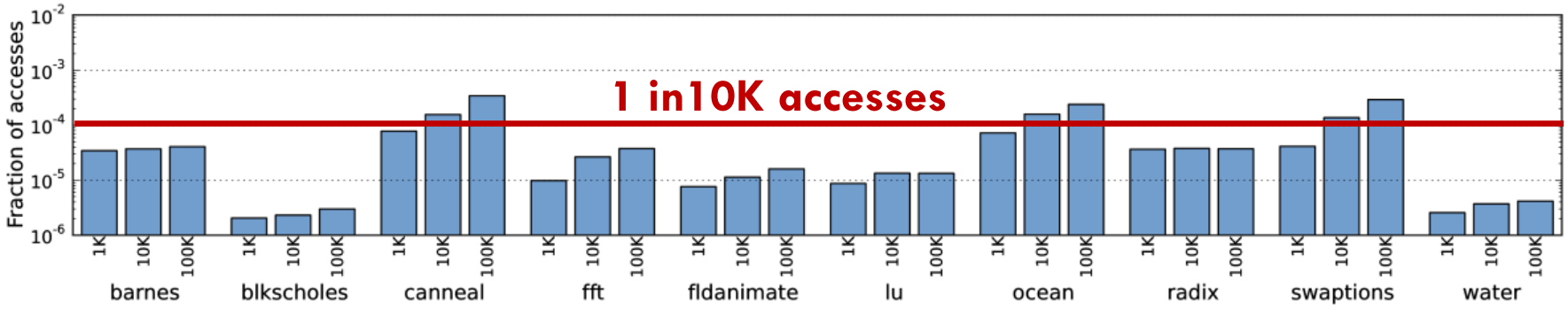
Path-preserving interference

If we simulate two accesses out of order, their timing changes but their paths do not



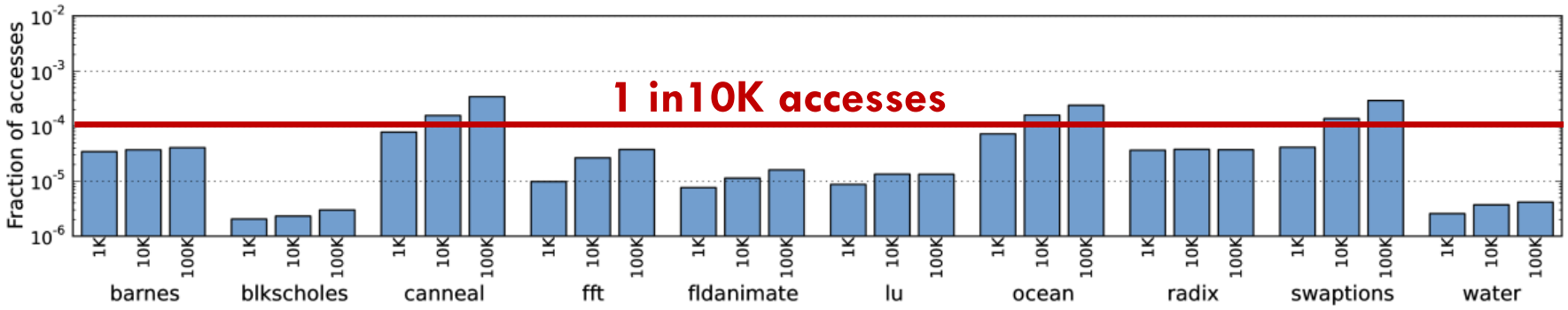
Characterizing Interference

- Accesses with path-altering interference with barrier synchronization every 1K/10K/100K cycles (64 cores):



Characterizing Interference

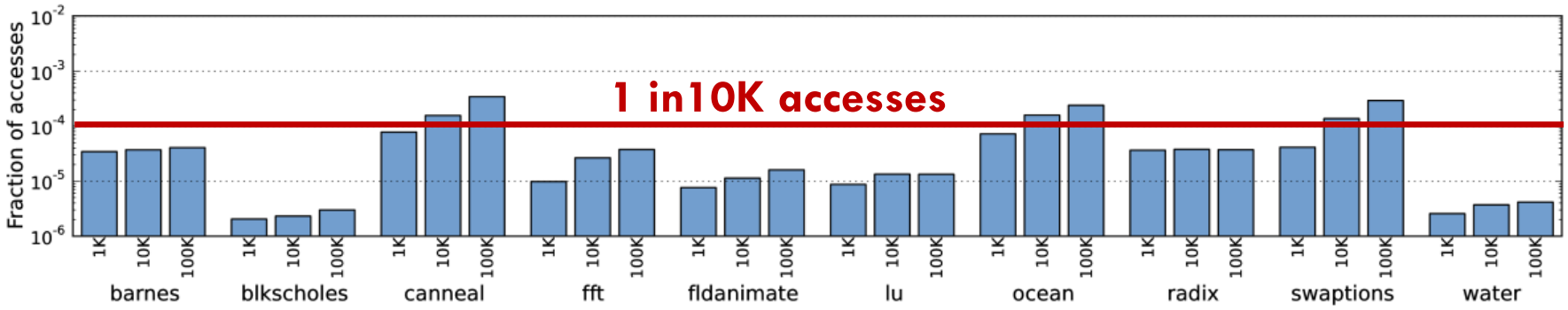
- Accesses with path-altering interference with barrier synchronization every 1K/10K/100K cycles (64 cores):



- Path-altering interference extremely rare in small intervals

Characterizing Interference

- Accesses with path-altering interference with barrier synchronization every 1K/10K/100K cycles (64 cores):



- Path-altering interference extremely rare in small intervals
- Strategy:
  - Simulate path-preserving interference faithfully
  - Ignore (but optionally profile) path-altering interference

Bound-Weave Parallelization

- Divide simulation in small intervals (e.g., 1000 cycles)
- Two parallel phases per interval: Bound and weave

Bound-Weave Parallelization

- Divide simulation in small intervals (e.g., 1000 cycles)
- Two parallel phases per interval: Bound and weave

Bound phase: Find paths

Weave phase: Find timings

Bound-Weave Parallelization

- Divide simulation in small intervals (e.g., 1000 cycles)
- Two parallel phases per interval: Bound and weave

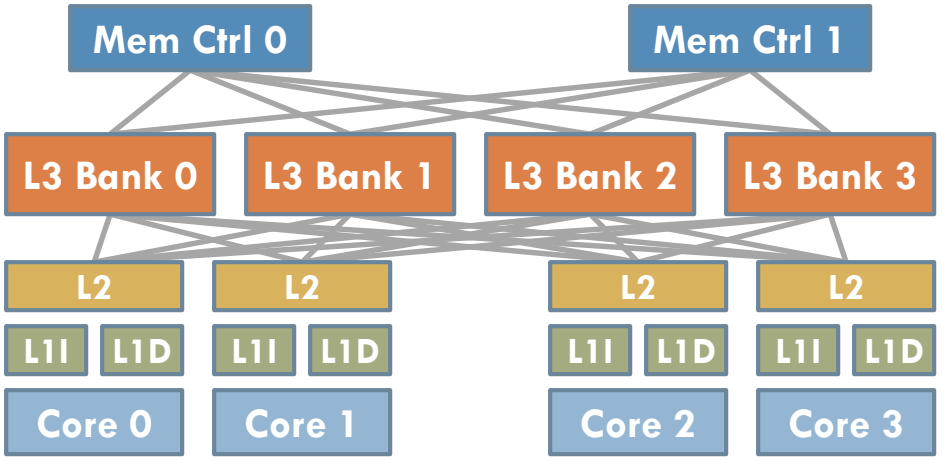
Bound phase: Find paths

Weave phase: Find timings

Bound-Weave equivalent to PDES
for path-preserving interference

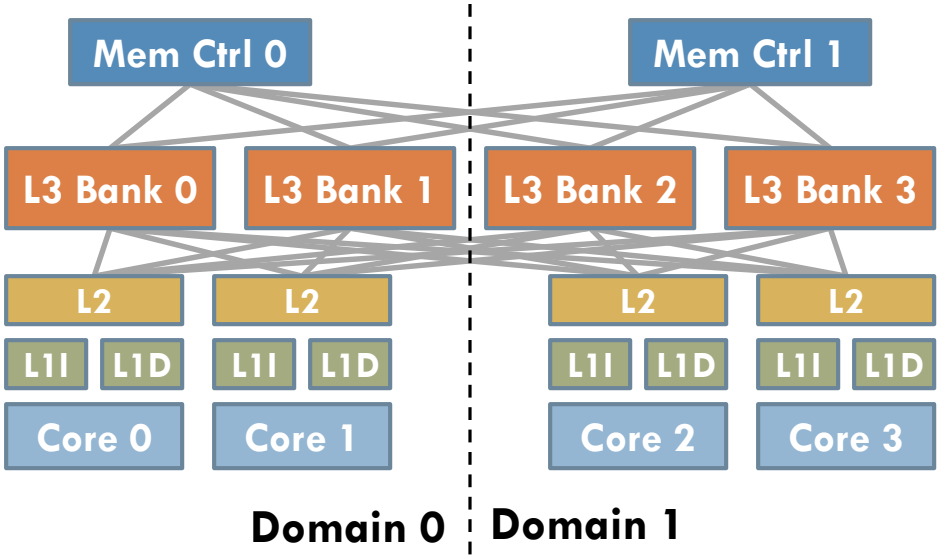
Bound-Weave Example

- 2-core host simulating 4-core system
- 1000-cycle intervals



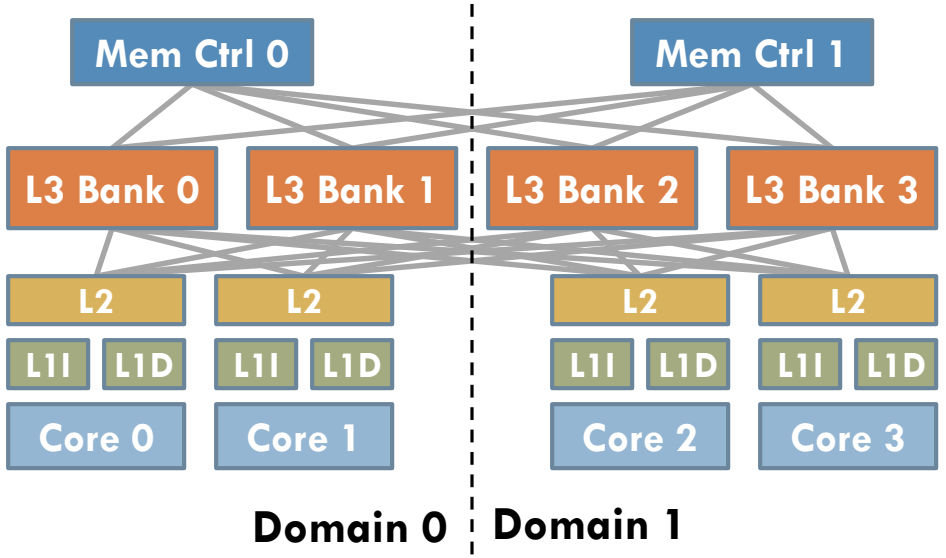
Bound-Weave Example

- 2-core host simulating 4-core system
- 1000-cycle intervals
- Divide components among 2 domains



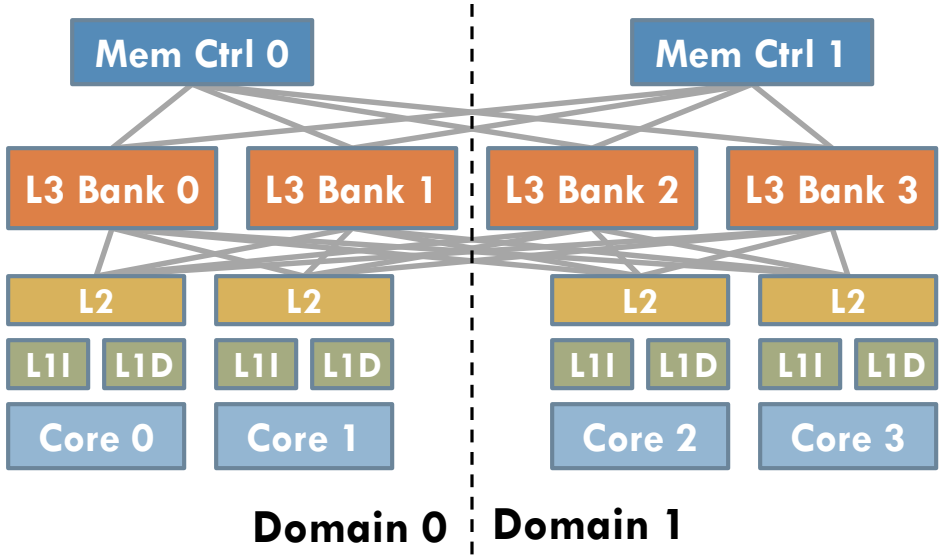
Bound-Weave Example

- 2-core host simulating 4-core system
- 1000-cycle intervals
- Divide components among 2 domains

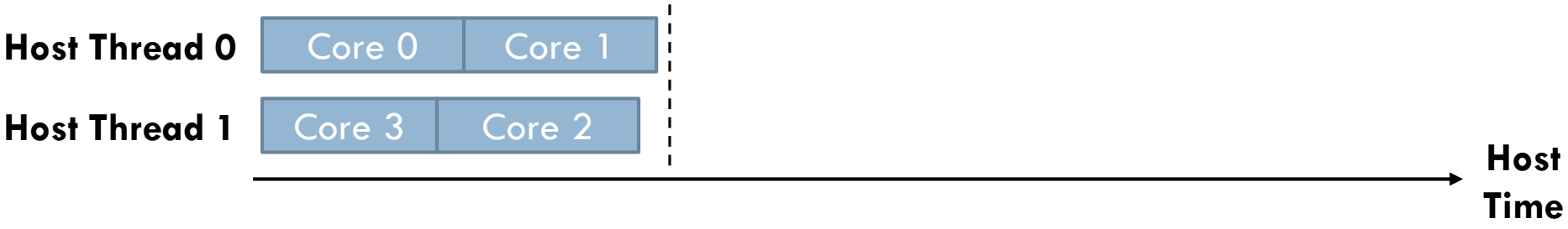


Bound-Weave Example

- 2-core host simulating 4-core system
- 1000-cycle intervals
- Divide components among 2 domains

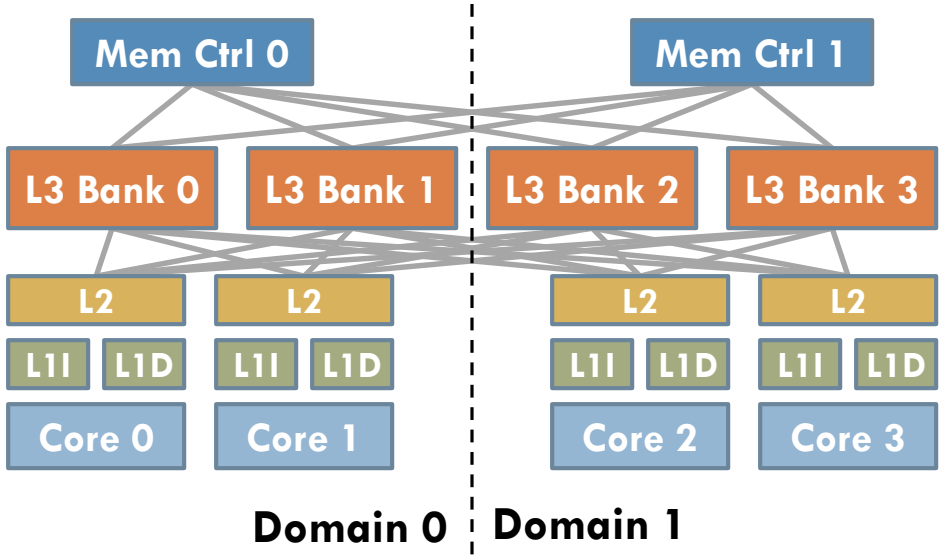


Bound Phase: Parallel simulation until cycle 1000, gather access traces

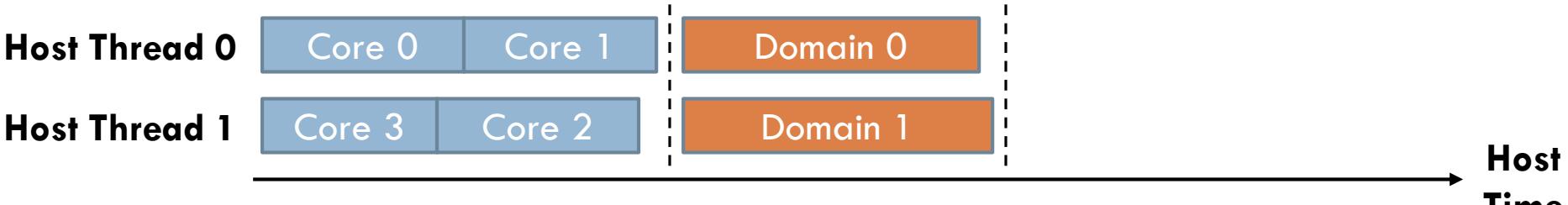


Bound-Weave Example

- 2-core host simulating 4-core system
- 1000-cycle intervals
- Divide components among 2 domains



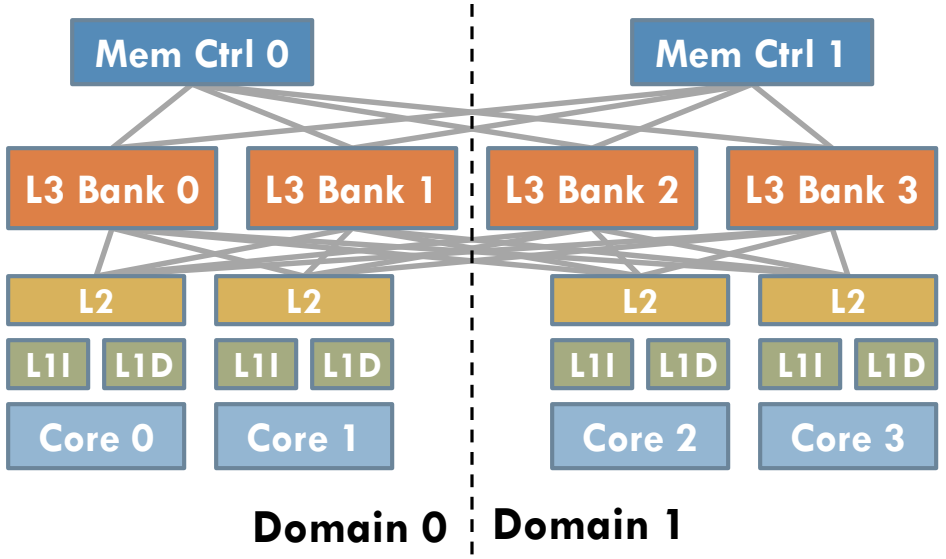
Bound Phase: Parallel simulation until cycle 1000, gather access traces



Weave Phase: Parallel event-driven simulation of gathered traces until actual cycle 1000

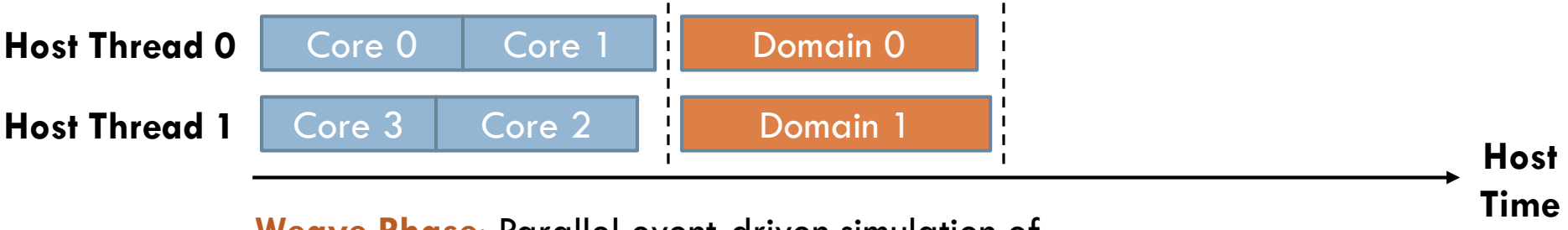
Bound-Weave Example

- 2-core host simulating 4-core system
- 1000-cycle intervals
- Divide components among 2 domains



Bound Phase: Parallel simulation until cycle 1000, gather access traces

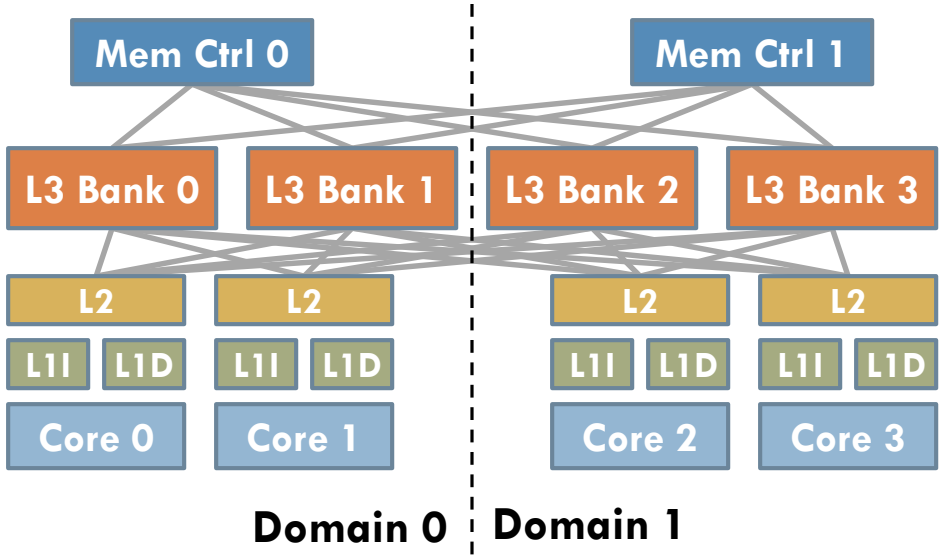
Feedback: Adjust core cycles



Weave Phase: Parallel event-driven simulation of gathered traces until actual cycle 1000

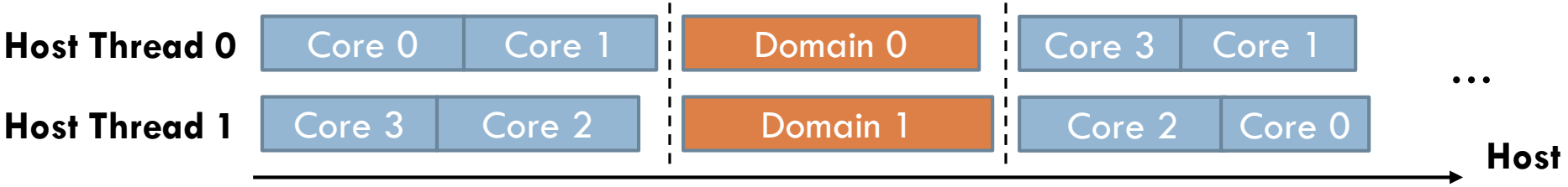
Bound-Weave Example

- 2-core host simulating 4-core system
- 1000-cycle intervals
- Divide components among 2 domains



Bound Phase: Parallel simulation until cycle 1000, gather access traces

Feedback: Adjust core cycles

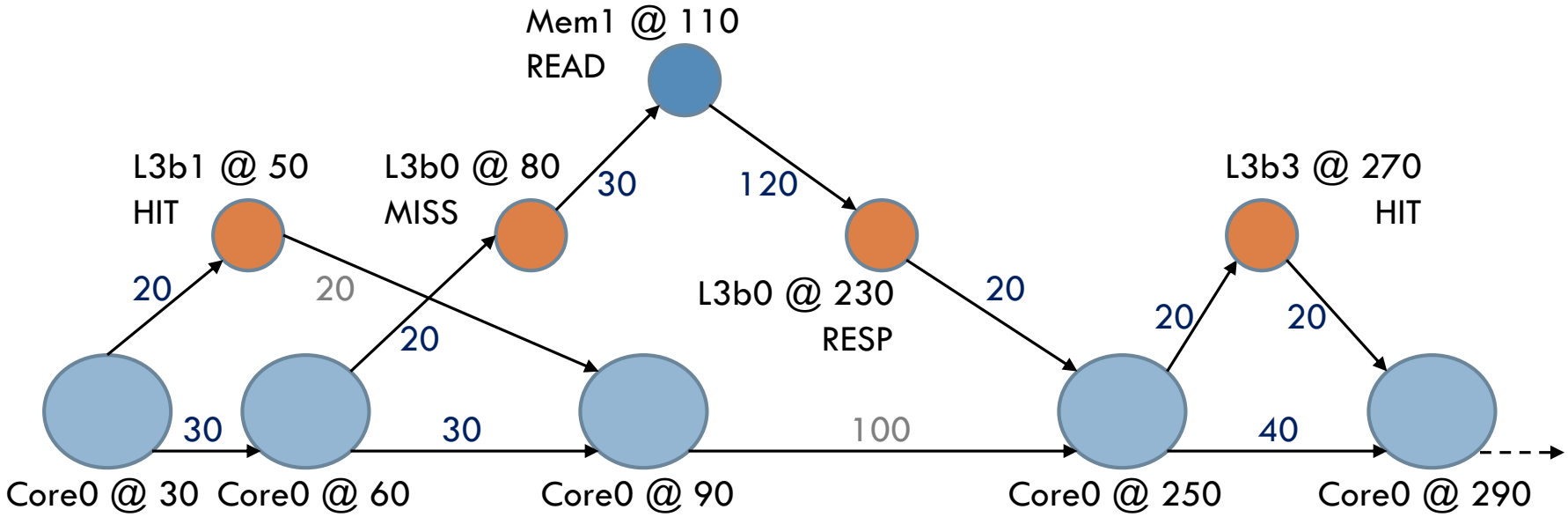


Weave Phase: Parallel event-driven simulation of gathered traces until actual cycle 1000

Bound Phase (until cycle 2000)

Example: Bound Phase

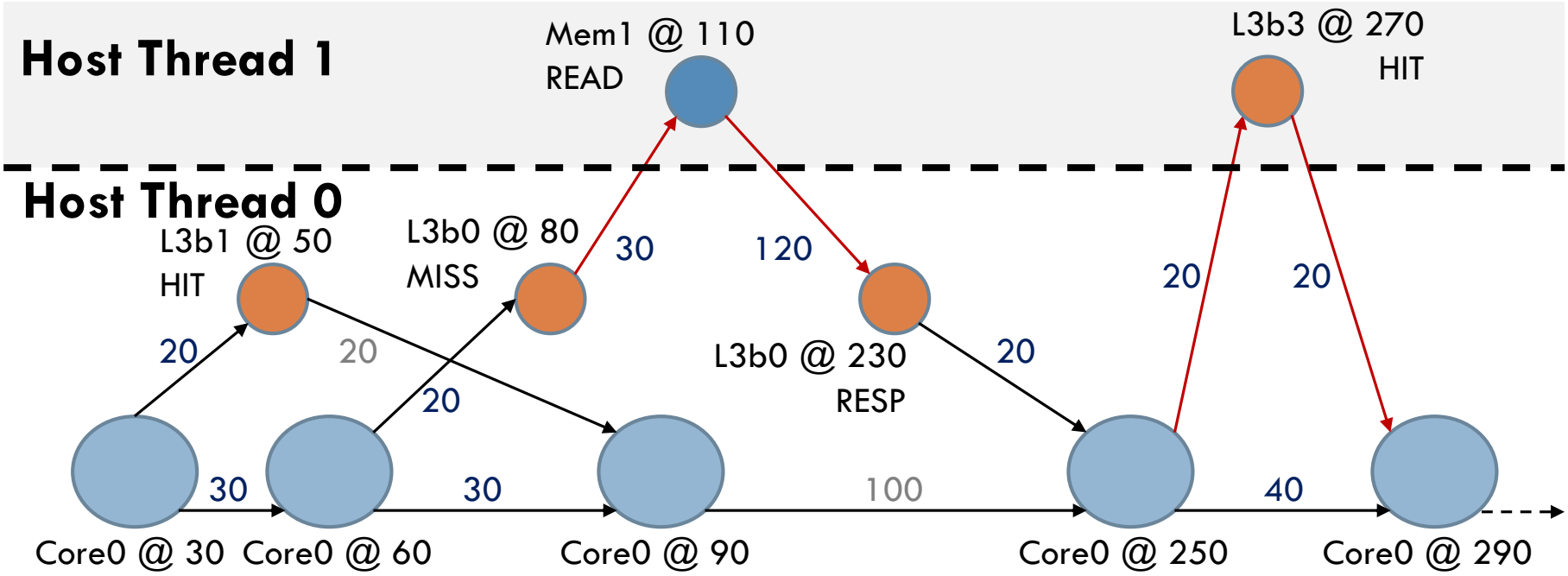
- Host thread 0 simulates core 0, records trace:



- Edges fix minimum latency between events
- Minimum L3 and main memory latencies (no interference)

Example: Weave Phase

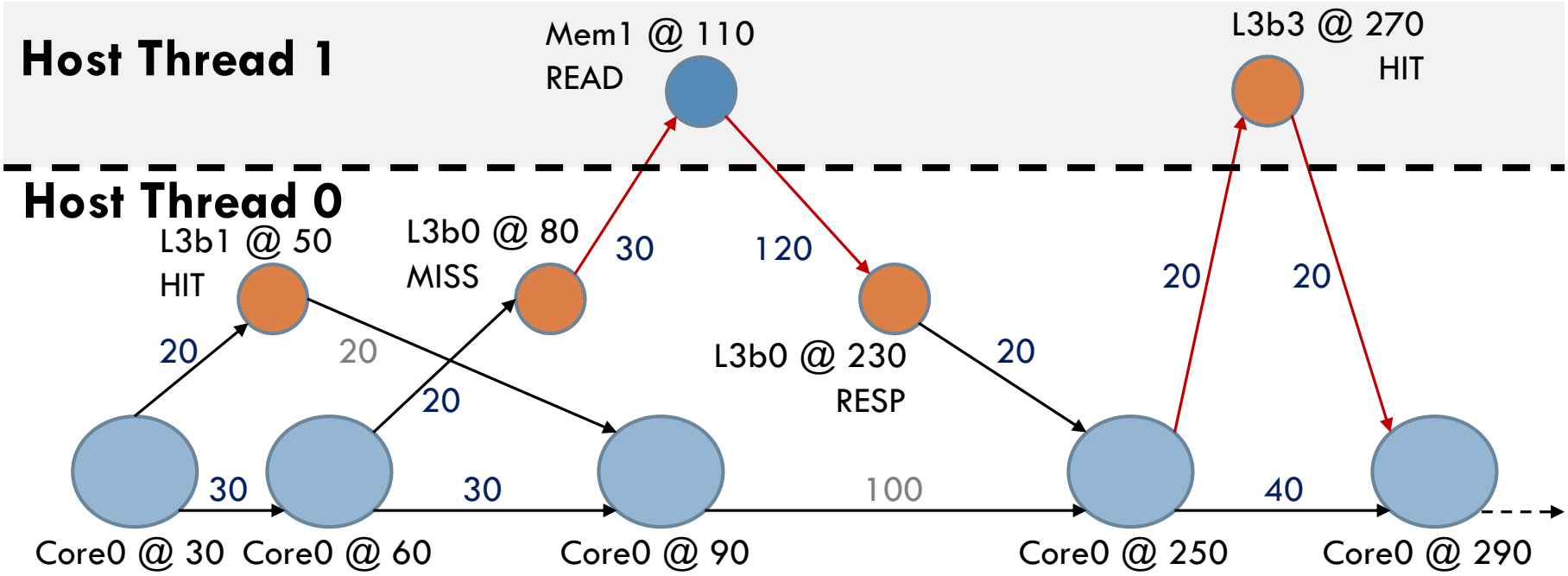
- Host threads simulate components from domains 0,1



- Host threads only sync when needed
  - e.g., thread 1 simulates other events (not shown) until cycle 110, syncs
  - Lower bounds guarantee no order violations

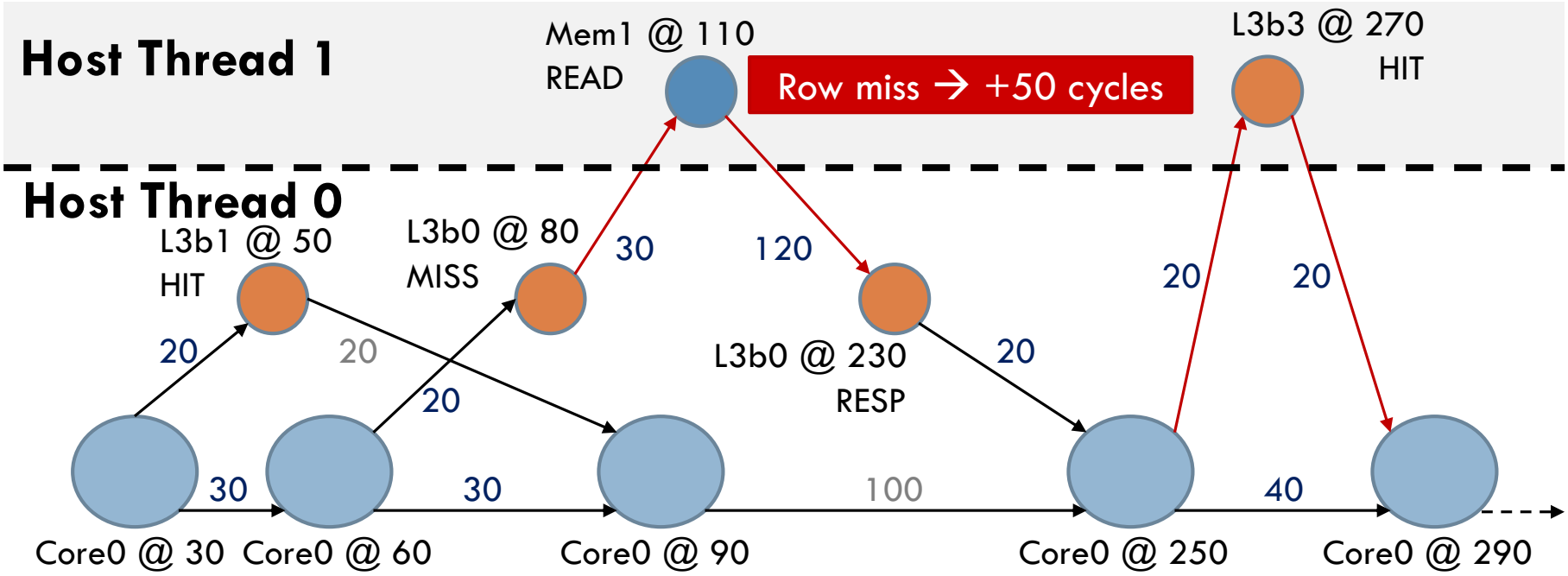
Example: Weave Phase

- Delays propagate as events are simulated:



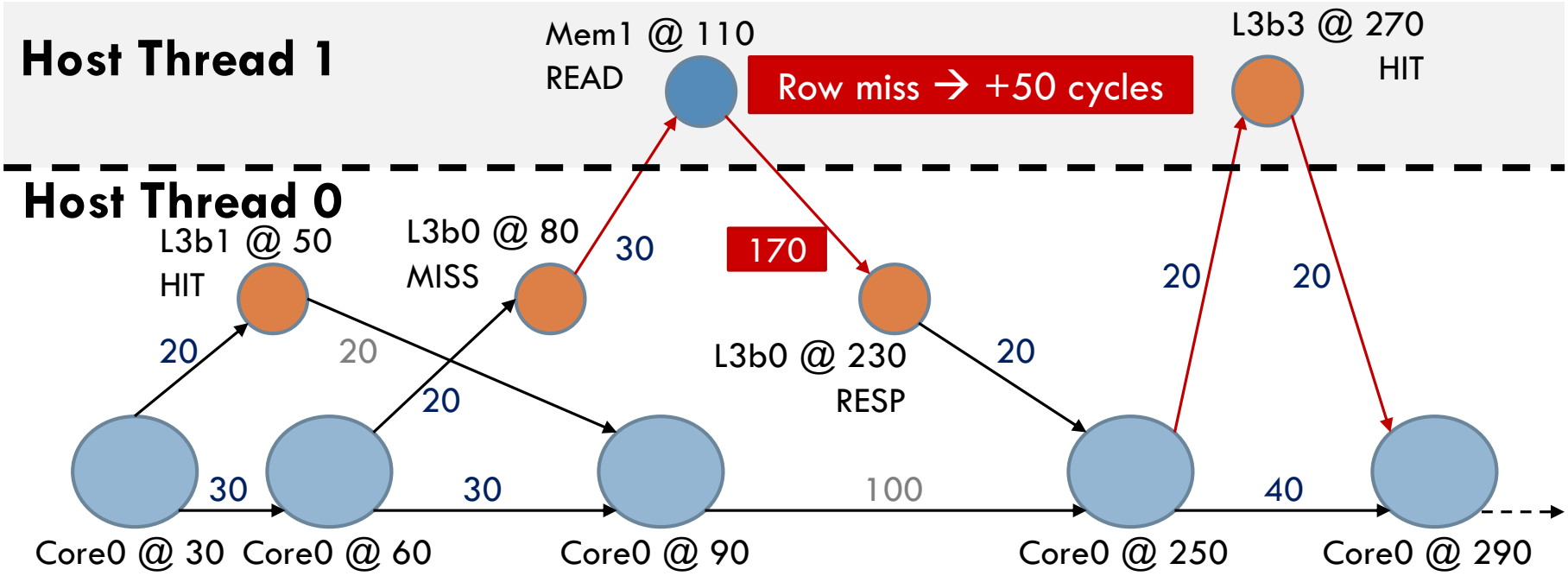
Example: Weave Phase

- Delays propagate as events are simulated:



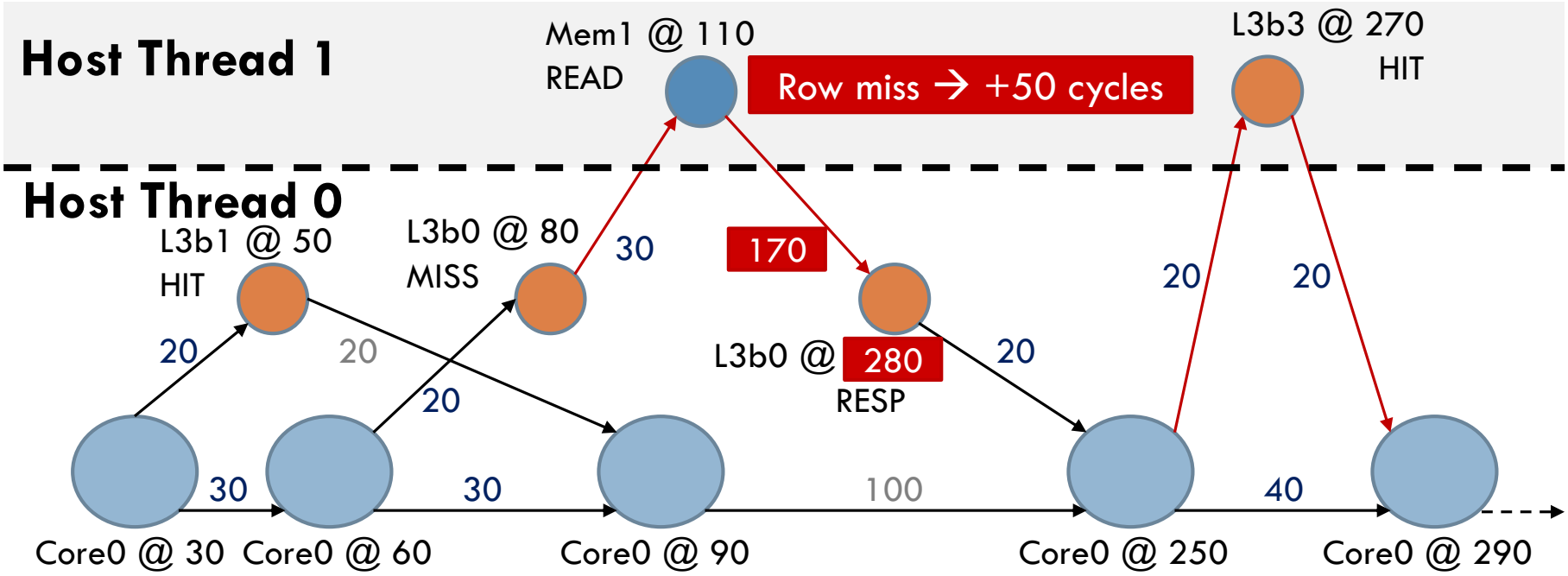
Example: Weave Phase

- Delays propagate as events are simulated:



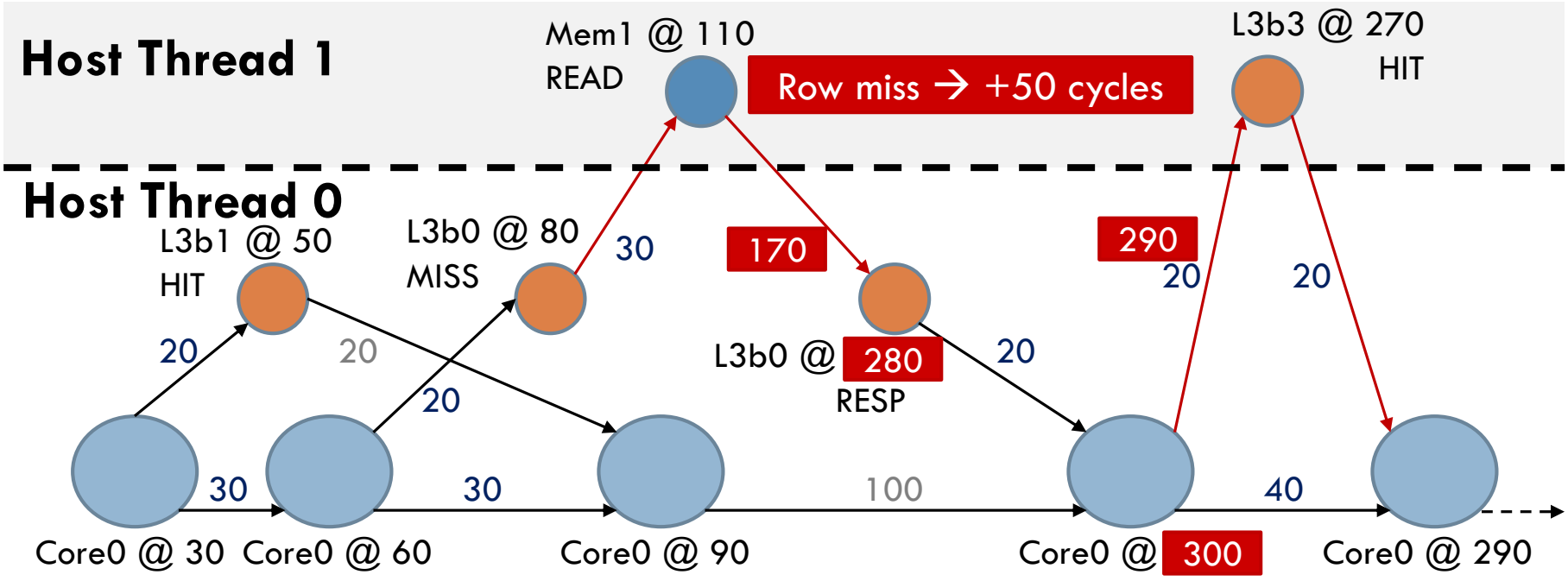
Example: Weave Phase

- Delays propagate as events are simulated:



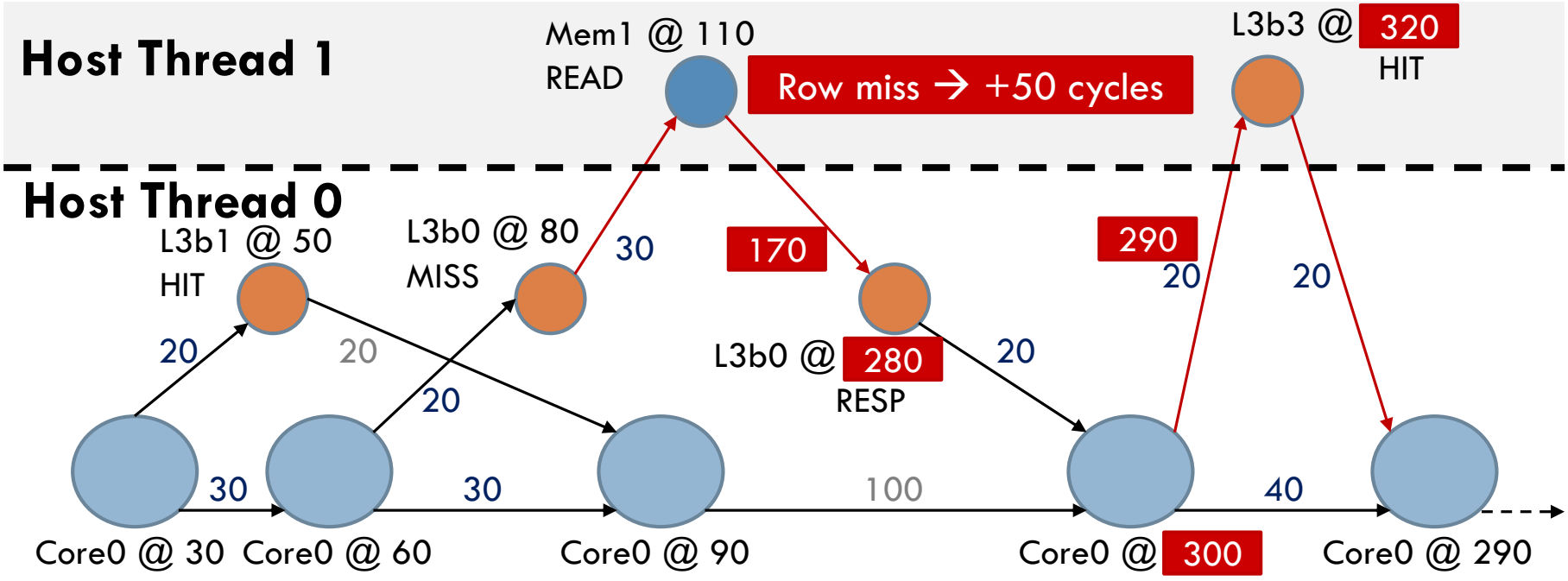
Example: Weave Phase

- Delays propagate as events are simulated:



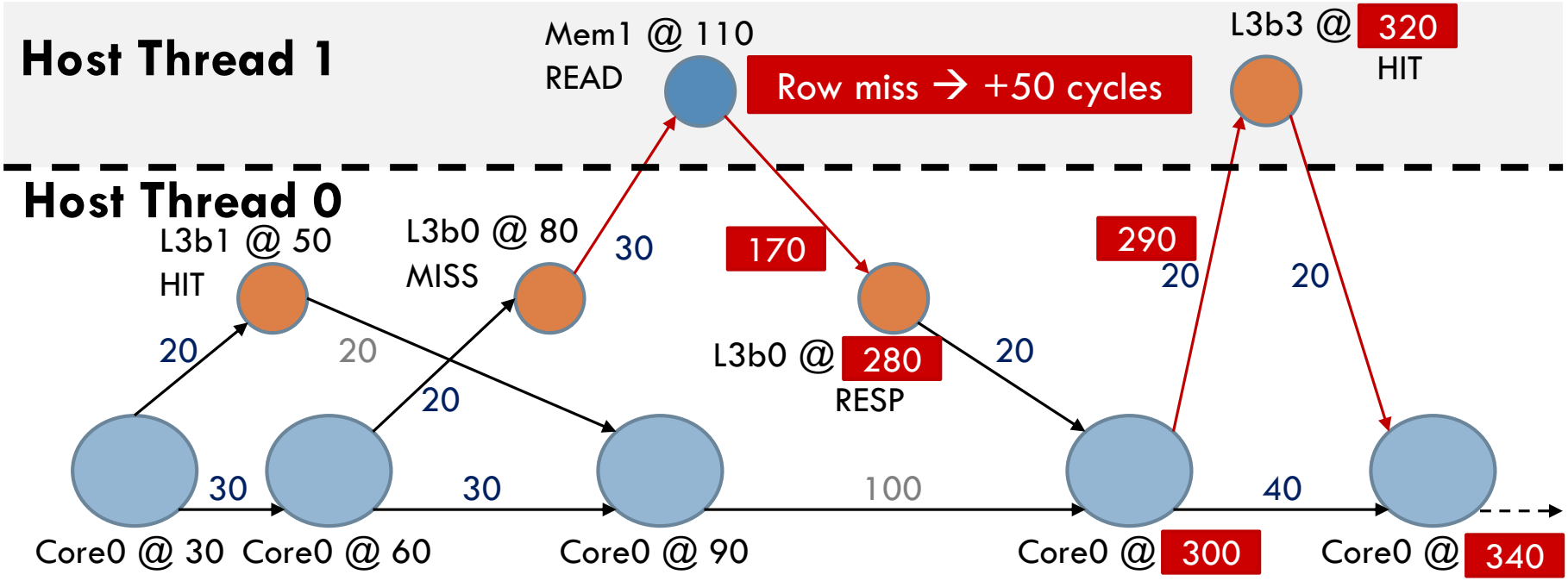
Example: Weave Phase

- Delays propagate as events are simulated:



Example: Weave Phase

- Delays propagate as events are simulated:



Bound-Weave Scalability

- Bound phase scales almost linearly
  - Using novel shared-memory synchronization protocol (later)
- Weave phase scales much better than PDES
  - Threads only need to sync when an event crosses domains
  - A lot of work shifted to bound phase

Bound-Weave Scalability

- Bound phase scales almost linearly
  - Using novel shared-memory synchronization protocol (later)
- Weave phase scales much better than PDES
  - Threads only need to sync when an event crosses domains
  - A lot of work shifted to bound phase
- Need bound and weave models for each component, but division is often very natural
  - e.g., caches: hit/miss on bound phase; MSHRs, pipelined accesses, port contention on weave phase

Bound-Weave Take-Aways

- Minimal synchronization:
  - Bound phase: Unordered accesses (like lax)
  - Weave: Only sync on actual dependencies

Bound-Weave Take-Aways

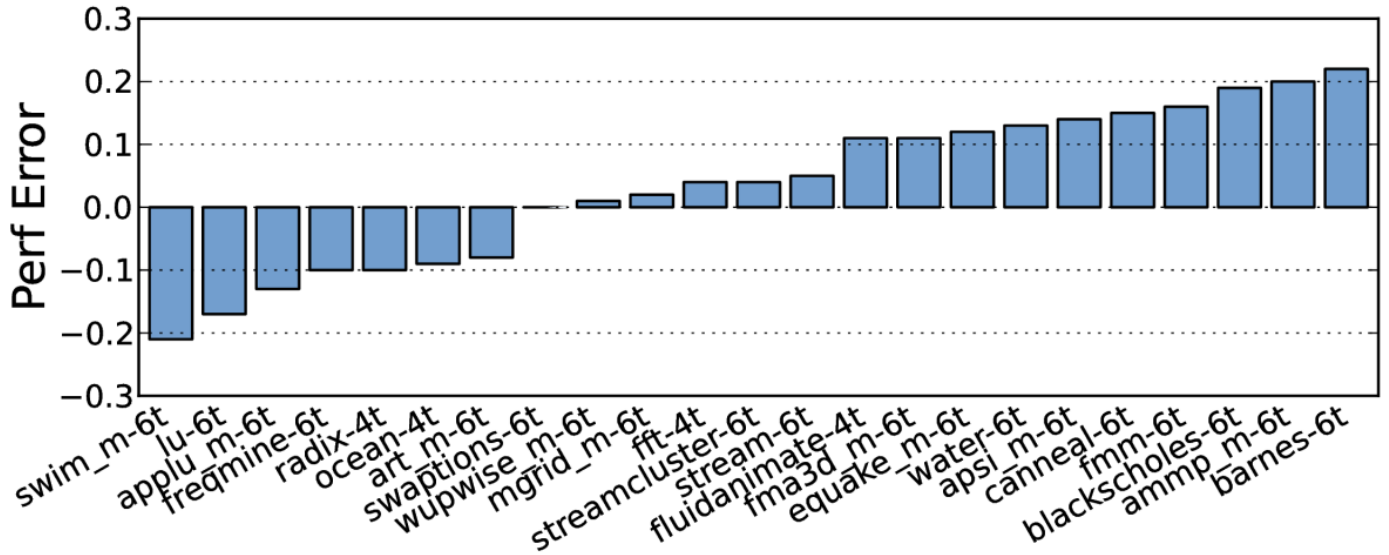
- Minimal synchronization:
  - Bound phase: Unordered accesses (like lax)
  - Weave: Only sync on actual dependencies
- No ordering violations in weave phase

Bound-Weave Take-Aways

- Minimal synchronization:
  - Bound phase: Unordered accesses (like lax)
  - Weave: Only sync on actual dependencies
- No ordering violations in weave phase
- Works with standard event-driven models
  - e.g., 110 lines to integrate with DRAMSim2

Multithreaded Accuracy

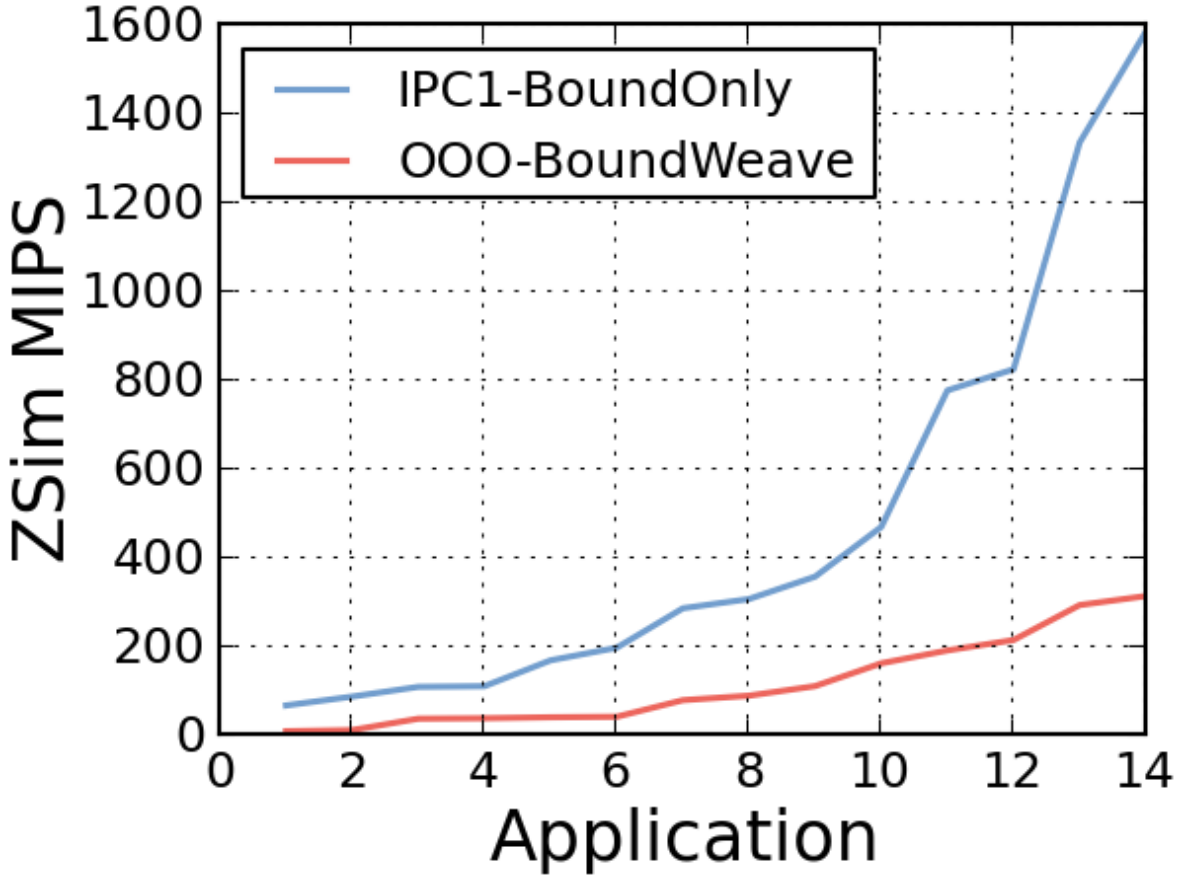
- 23 apps: PARSEC, SPLASH-2, SPEC OMP2001, STREAM



- 11.2% avg perf error (not IPC), 10/23 within 10%
  - Similar differences as single-core results

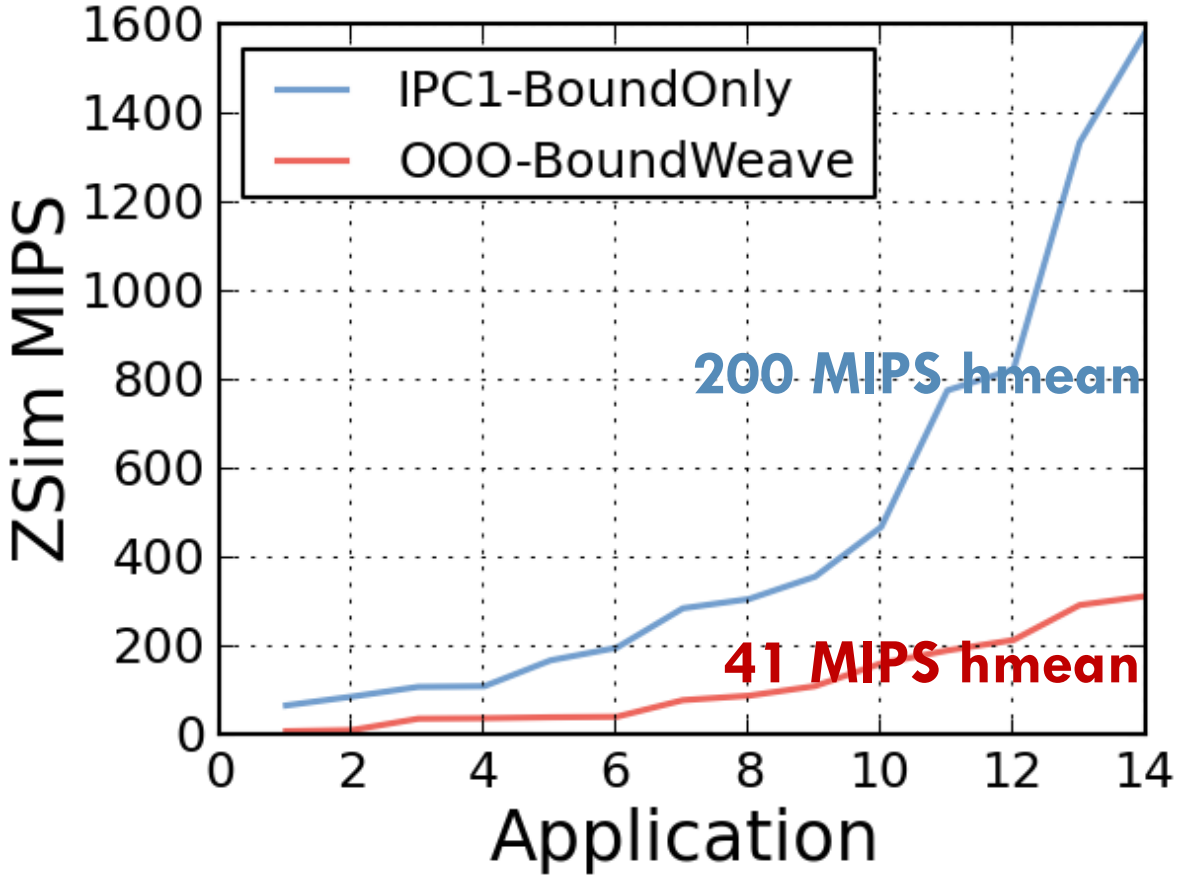
1024-Core Performance

- Host: 2-socket Sandy Bridge @ 2.6 GHz (16 cores, 32 threads)
- Results for the 14/23 parallel apps that scale



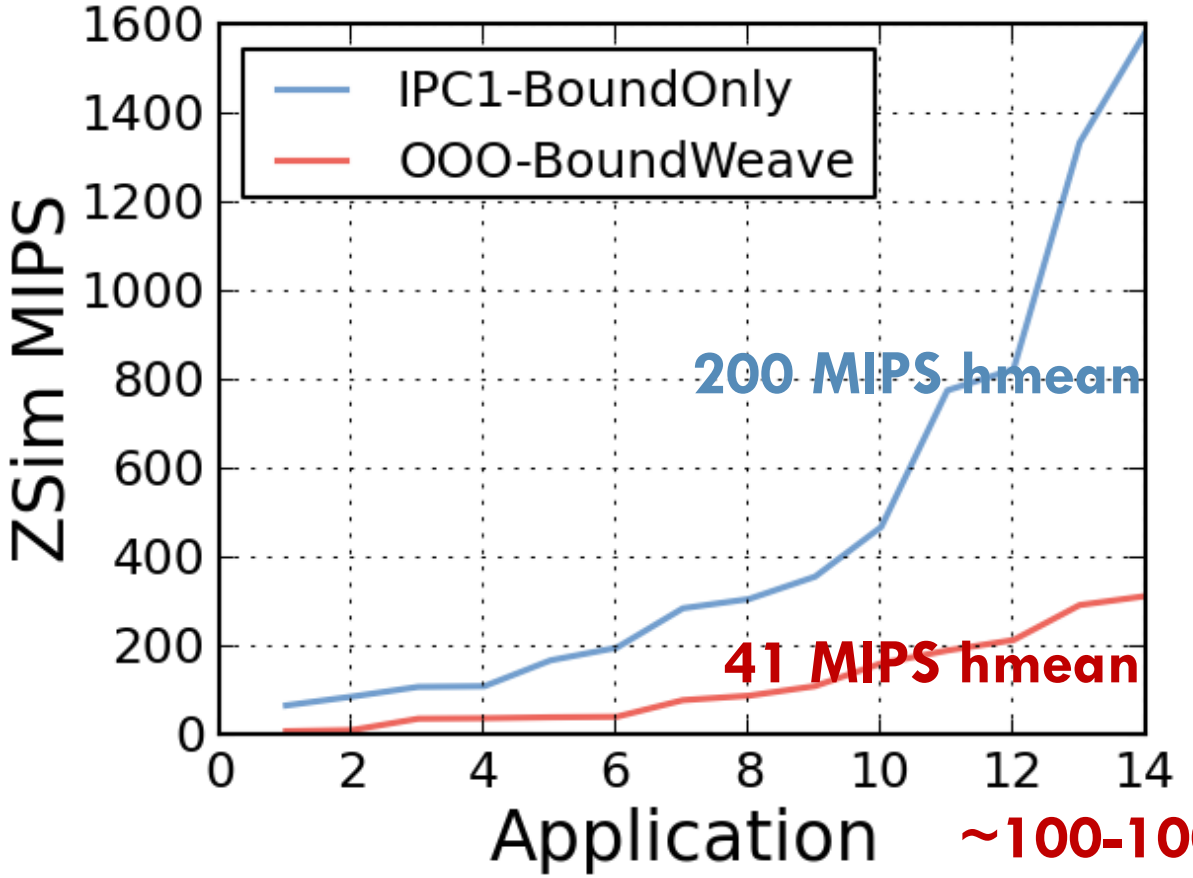
1024-Core Performance

- Host: 2-socket Sandy Bridge @ 2.6 GHz (16 cores, 32 threads)
- Results for the 14/23 parallel apps that scale



1024-Core Performance

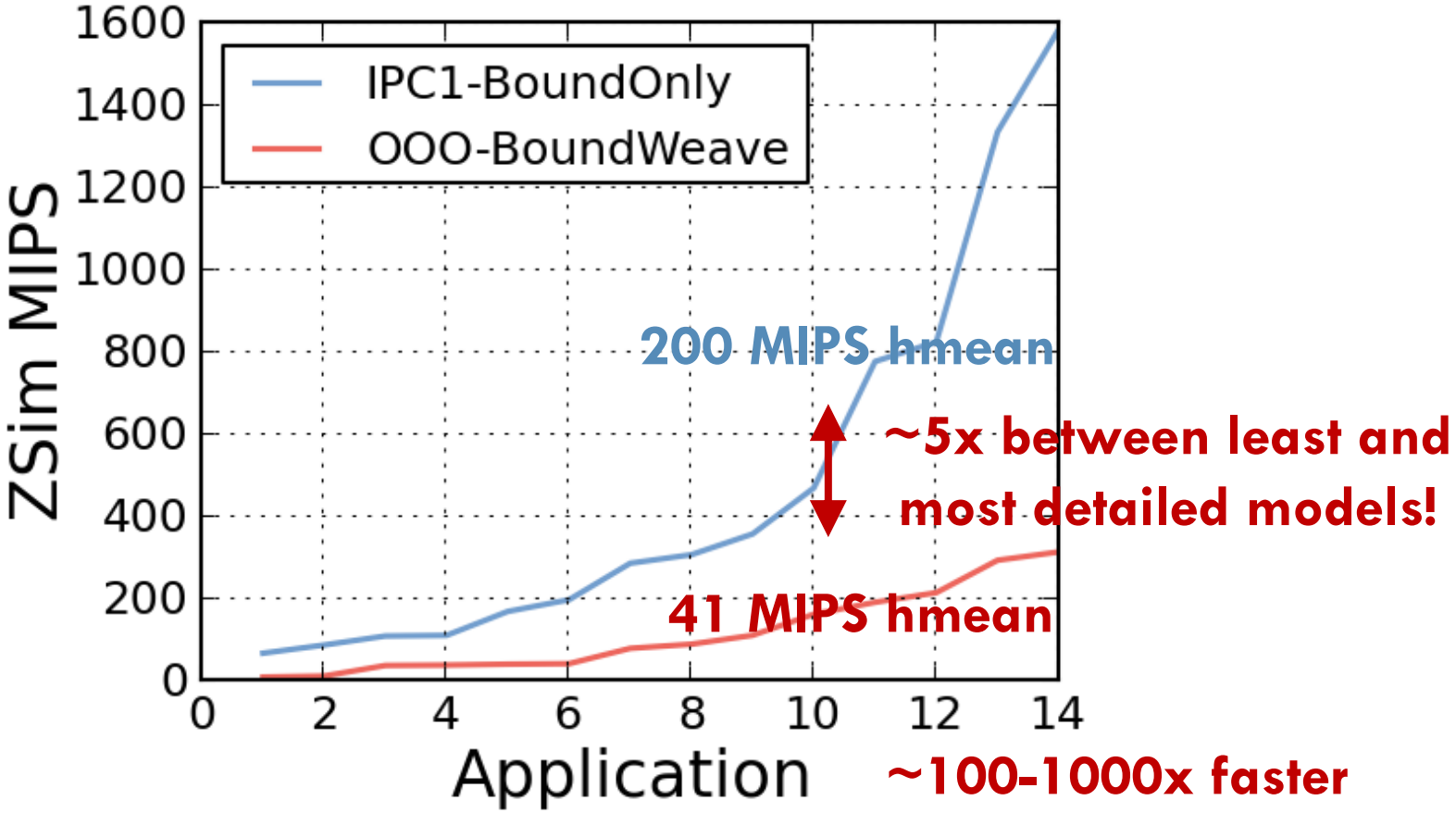
- Host: 2-socket Sandy Bridge @ 2.6 GHz (16 cores, 32 threads)
- Results for the 14/23 parallel apps that scale



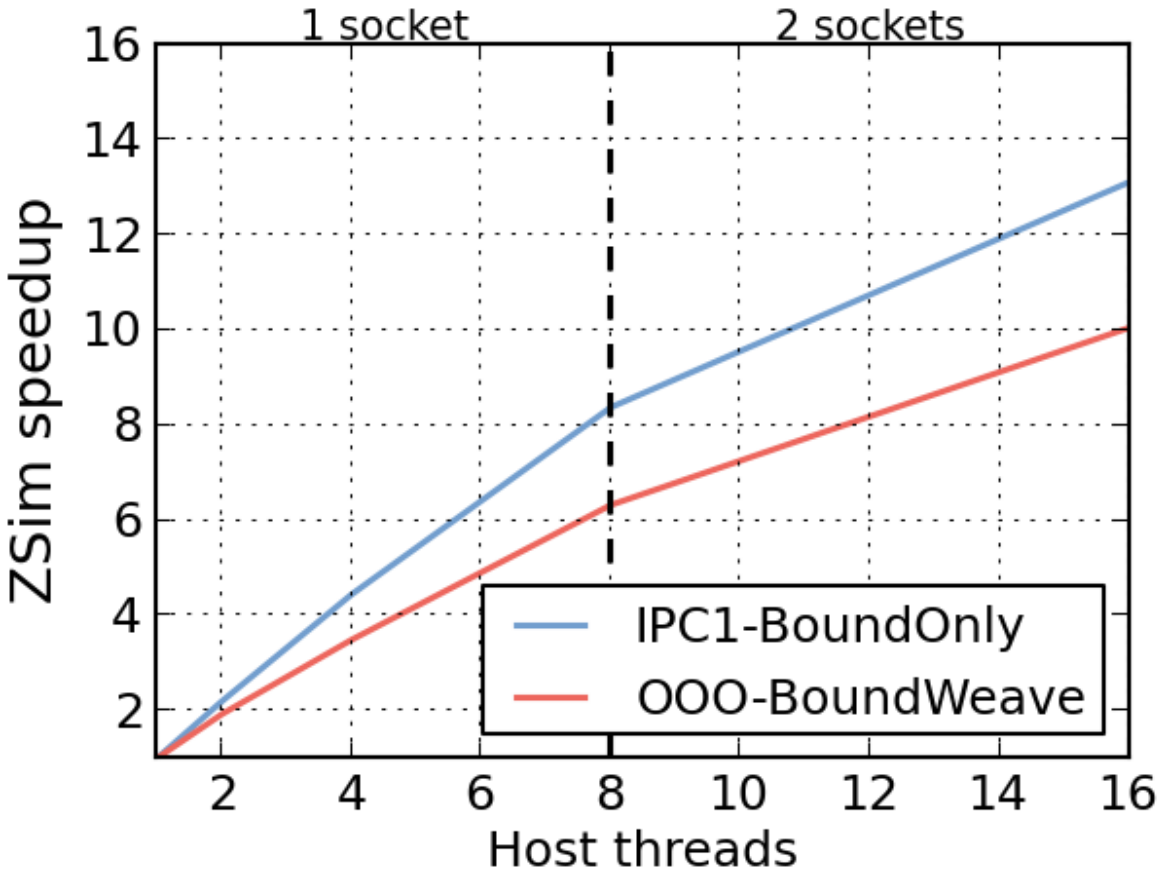
~100-1000x faster

1024-Core Performance

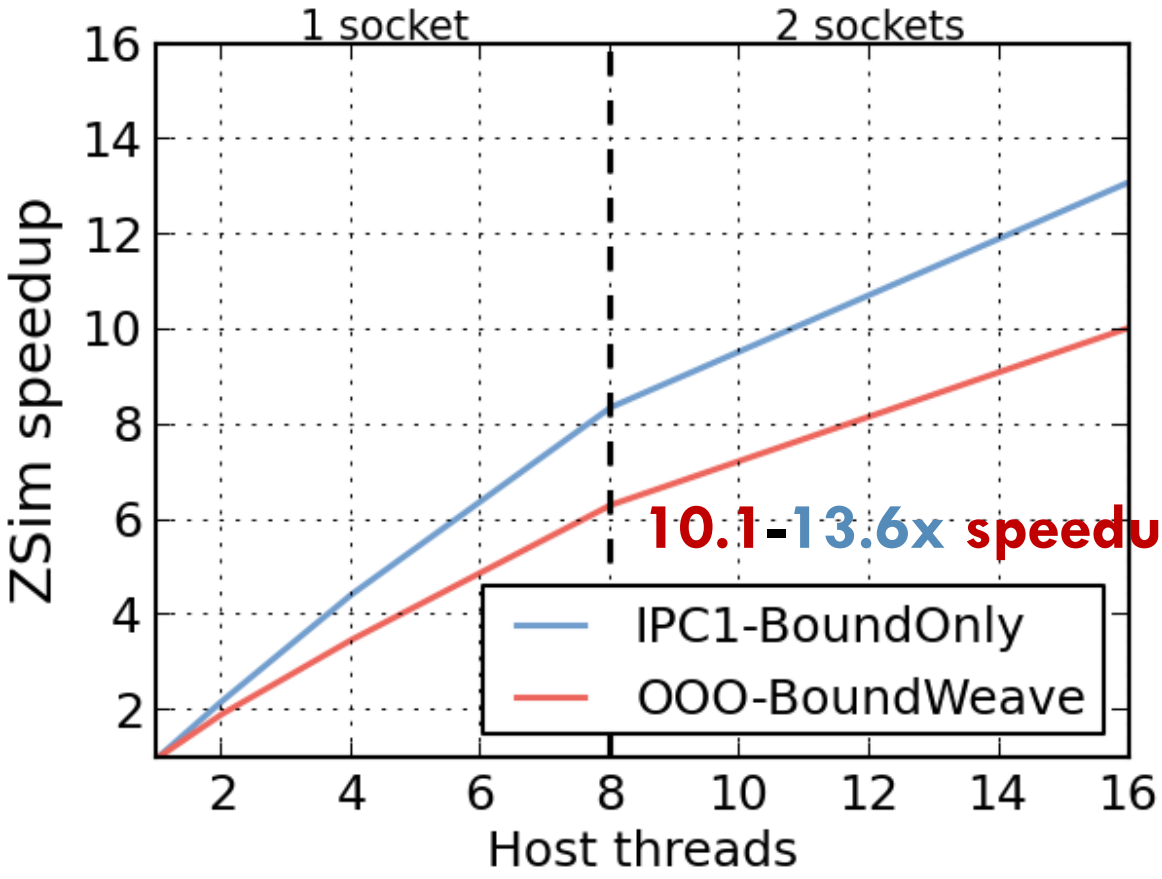
- Host: 2-socket Sandy Bridge @ 2.6 GHz (16 cores, 32 threads)
- Results for the 14/23 parallel apps that scale



Bound-Weave Scalability



Bound-Weave Scalability



10.1-13.6x speedup @ 16 cores

Outline

- Introduction
- Detailed DBT-accelerated core models
- Bound-weave parallelization
- Lightweight user-level virtualization

Lightweight User-Level Virtualization

- No 1Kcore OSs
- No parallel full-system DBT

} ZSim has to be user-level for now

Lightweight User-Level Virtualization

- No 1Kcore OSs
- No parallel full-system DBT

} ZSim has to be user-level for now

- Problem: User-level simulators limited to simple workloads

- Lightweight user-level virtualization: Bridge the gap with full-system simulation
  - Simulate accurately if time spent in OS is minimal

Lightweight User-Level Virtualization

- Multiprocess workloads
- Scheduler (threads > cores)
- Time virtualization
- System virtualization

- Simulator-OS deadlock avoidance
- Signals
- ISA extensions
- Fast-forwarding

ZSim Limitations

- Not implemented yet:
  - Multithreaded cores
  - Detailed NoC models
  - Virtual memory (TLBs)

ZSim Limitations

- Not implemented yet:
  - Multithreaded cores
  - Detailed NoC models
  - Virtual memory (TLBs)

- Fundamentally hard:
  - Systems or workloads with frequent path-altering interference (e.g., fine-grained message-passing across whole chip)
  - Kernel-intensive applications

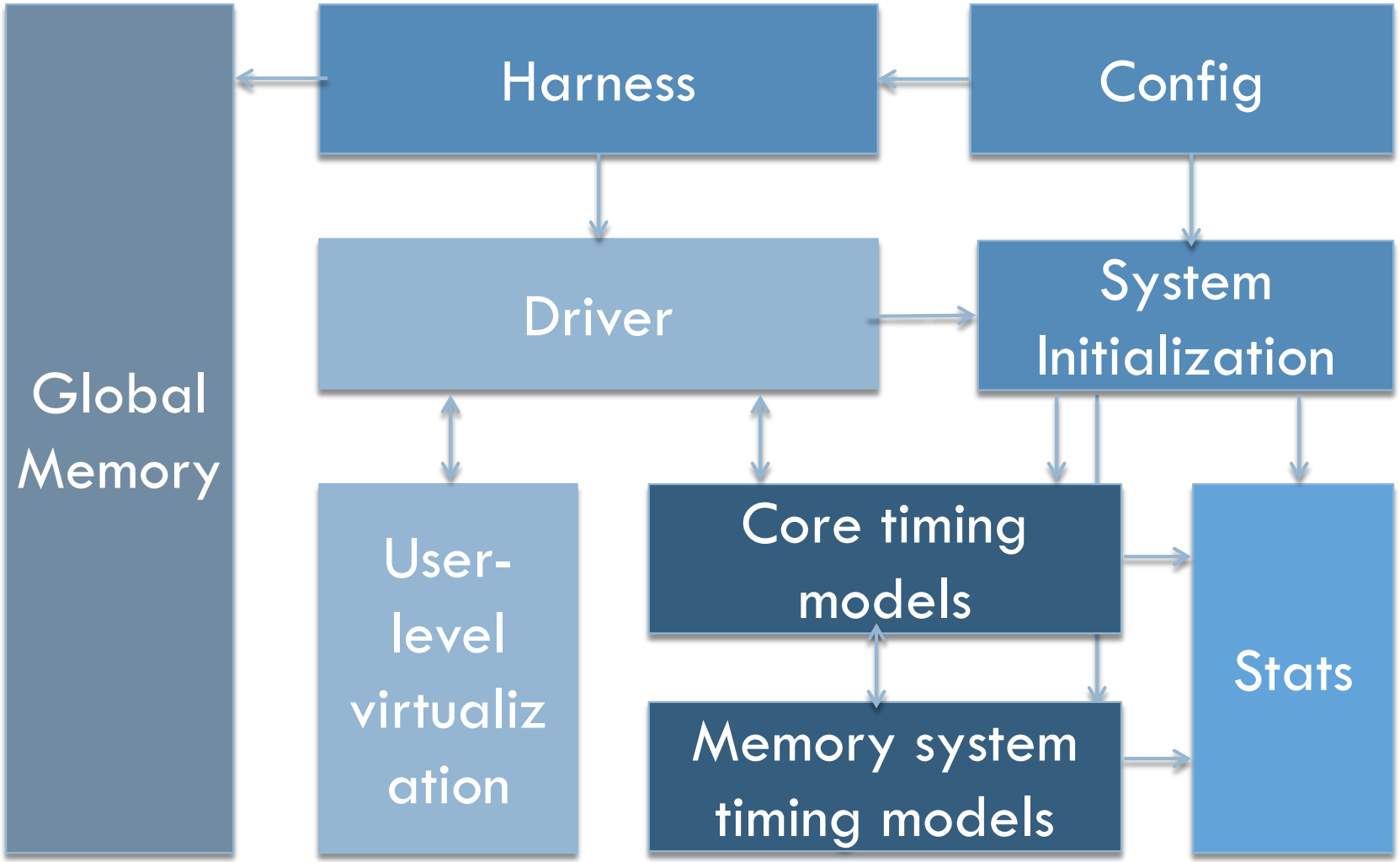
Summary

- Three techniques to make 1Kcore simulation practical
  - DBT-accelerated models: 10-100x faster core models
  - Bound-weave parallelization: ~10-15x speedup from parallelization with minimal accuracy loss
  - Lightweight user-level virtualization: Simulate complex workloads without full-system support
- ZSim achieves high performance and accuracy:
  - Simulates 1024-core systems at 10s-1000s of MIPS
  - Validated against real Westmere system, avg error ~10%

Simulator Organization

Massachusetts Institute of Technology

Main Components



ZSim Harness

- Most of zsim implemented as a pintool (libzsim.so)
- A separate harness process (zsim) controls simulation
  - Initializes global memory
  - Launches pin processes
  - Checks for deadlock

ZSim Harness

- Most of zsim implemented as a pintool (libzsim.so)
- A separate harness process (zsim) controls simulation
  - Initializes global memory
  - Launches pin processes
  - Checks for deadlock

```
./build/opt/zsim test.cfg
```

ZSim Harness

- Most of zsim implemented as a pintool (libzsim.so)
- A separate harness process (zsim) controls simulation
  - Initializes global memory
  - Launches pin processes
  - Checks for deadlock

```
./build/opt/zsim test.cfg
```

ZSim Harness

- Most of zsim implemented as a pintool (libzsim.so)
- A separate harness process (zsim) controls simulation
  - Initializes global memory
  - Launches pin processes
  - Checks for deadlock

```
./build/opt/zsim test.cfg
```

ZSim Harness

- Most of zsim implemented as a pintool (libzsim.so)
- A separate harness process (zsim) controls simulation
  - Initializes global memory
  - Launches pin processes
  - Checks for deadlock

```
./build/opt/zsim test.cfg
```

ZSim Harness

- Most of zsim implemented as a pintool (libzsim.so)
- A separate harness process (zsim) controls simulation
  - Initializes global memory
  - Launches pin processes
  - Checks for deadlock

```
./build/opt/zsim test.cfg
```

```
process0 = {
  command = "ls";
};

process1 = {
   command = "echo foo";
};

…
```

ZSim Harness

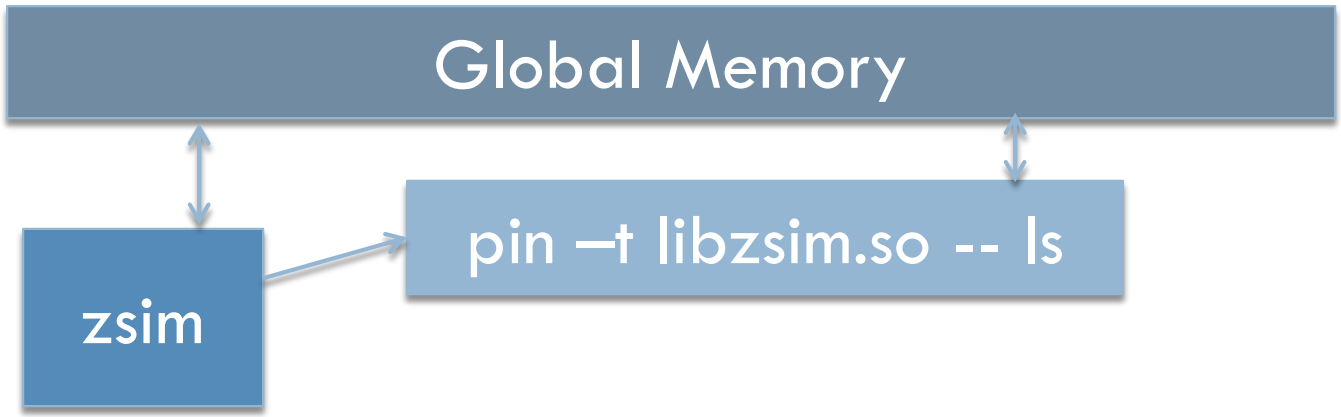
- Most of zsim implemented as a pintool (libzsim.so)
- A separate harness process (zsim) controls simulation
  - Initializes global memory
  - Launches pin processes
  - Checks for deadlock

```
./build/opt/zsim test.cfg

    process0 = {
      command = "ls";
    };

    process1 = {
       command = "echo foo";
    };

    …
```



ZSim Harness

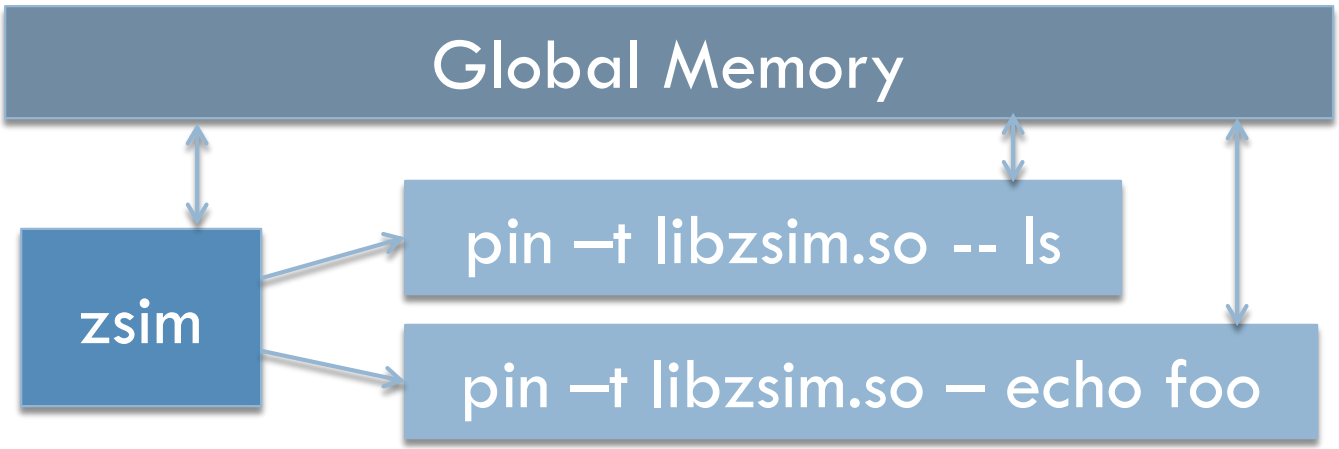
- Most of zsim implemented as a pintool (libzsim.so)
- A separate harness process (zsim) controls simulation
  - Initializes global memory
  - Launches pin processes
  - Checks for deadlock

```
./build/opt/zsim test.cfg

    process0 = {
      command = "ls";
    };

    process1 = {
       command = "echo foo";
    };

    …
```

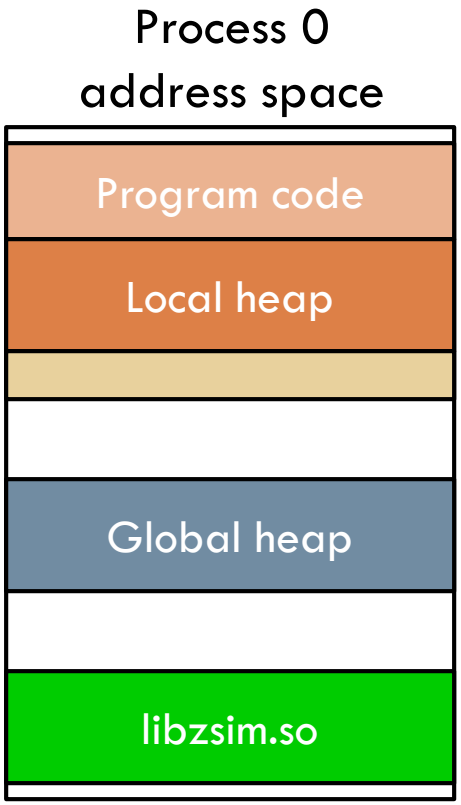


Global Memory

- Pin processes communicate through a shared memory segment, managed as a single global heap
- All simulator objects must be allocated in the global heap

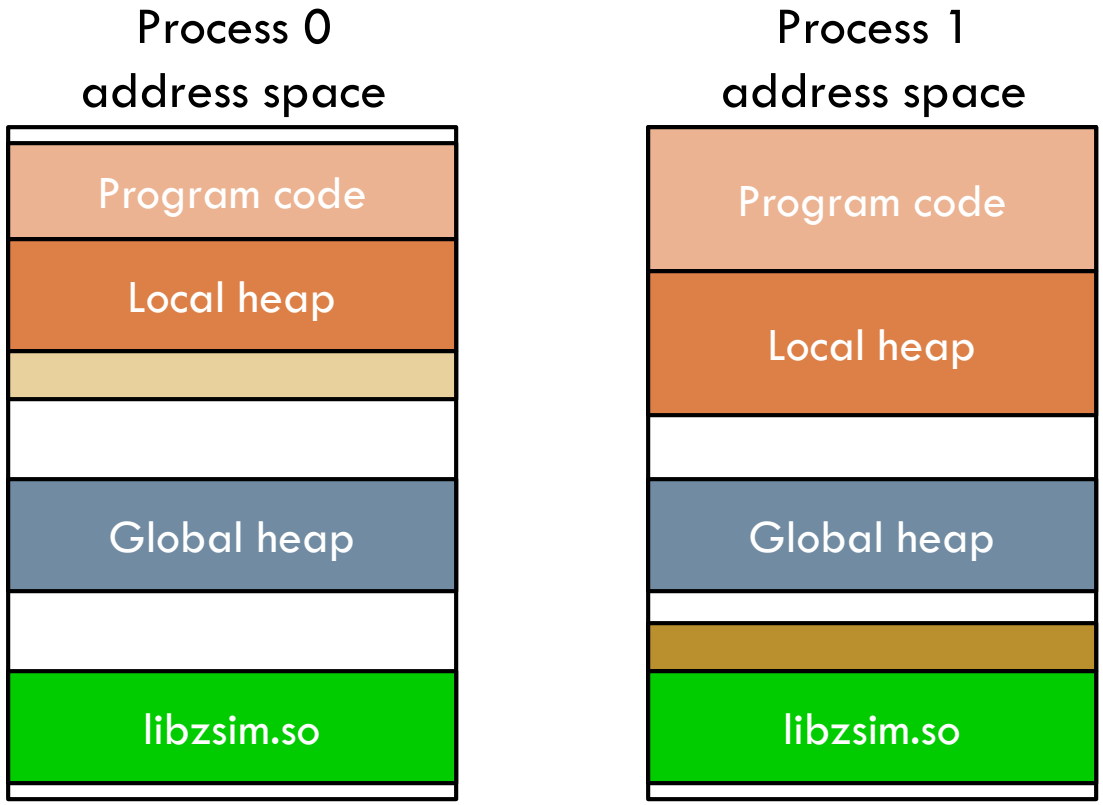
Global Memory

- Pin processes communicate through a shared memory segment, managed as a single global heap
- All simulator objects must be allocated in the global heap



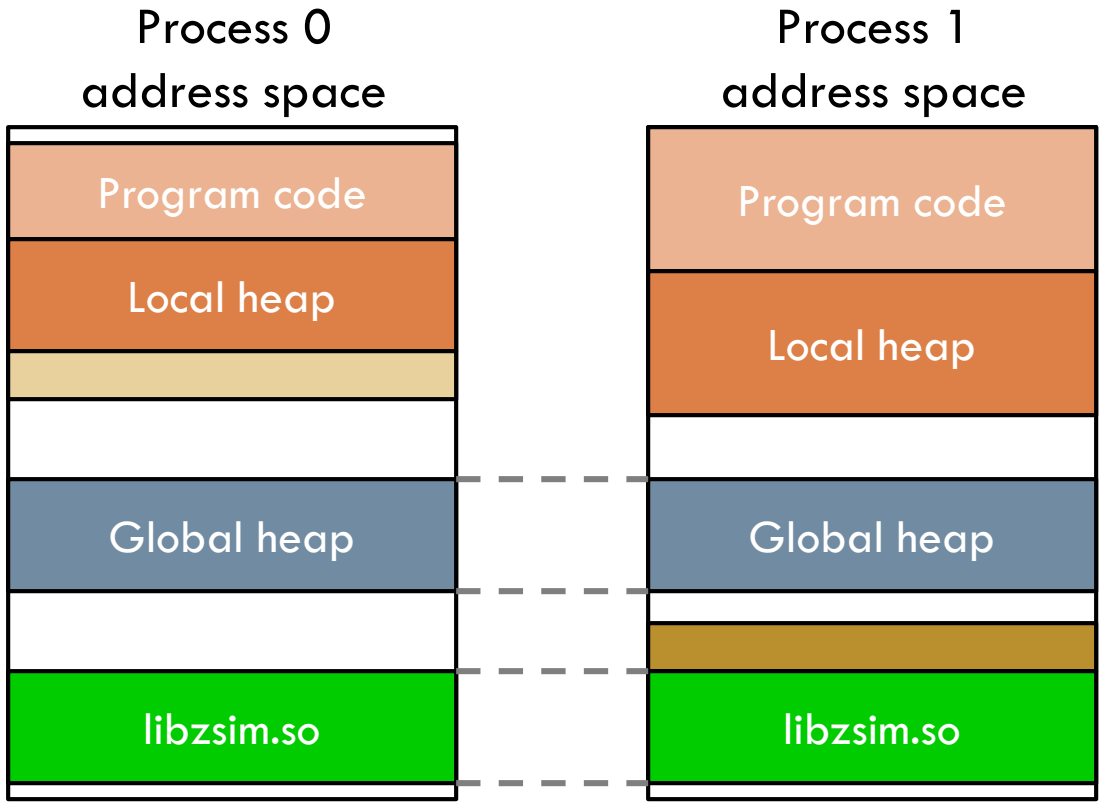
Global Memory

- Pin processes communicate through a shared memory segment, managed as a single global heap
- All simulator objects must be allocated in the global heap



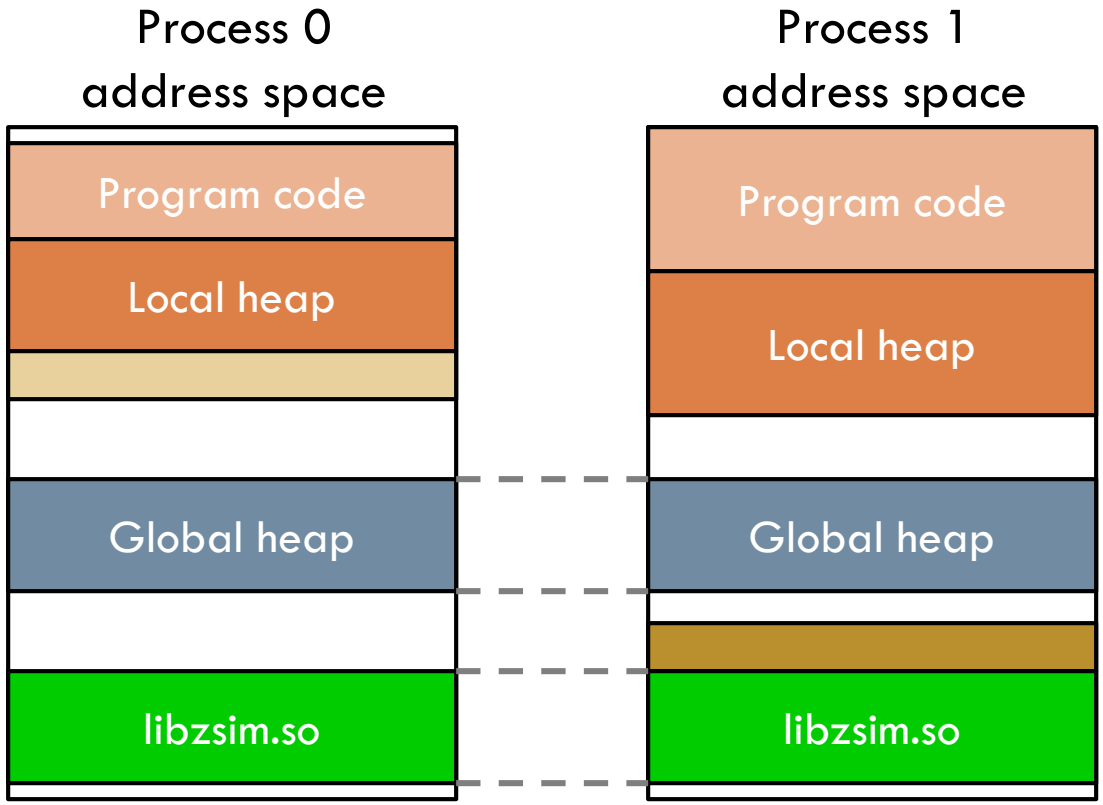
Global Memory

- Pin processes communicate through a shared memory segment, managed as a single global heap
- All simulator objects must be allocated in the global heap



Global Memory

- Pin processes communicate through a shared memory segment, managed as a single global heap
- All simulator objects must be allocated in the global heap



Global heap and libzsim.so code in same memory locations across all processes → Can use normal pointers & virtual functions

Global Memory Allocation Idioms

- Globally-allocated objects: Inherit from GlobAlloc

```
class SimObject : GlobAlloc { …
```

Global Memory Allocation Idioms

- Globally-allocated objects: Inherit from GlobAlloc

```
class SimObject : GlobAlloc { …
```

- STL classes that allocate heap memory: Use g_stl variants

```
g_vector<uint64_t> cacheLines;
```

Global Memory Allocation Idioms

- Globally-allocated objects: Inherit from GlobAlloc

```
class SimObject : GlobAlloc { …
```

- STL classes that allocate heap memory: Use g_stl variants

```
g_vector<uint64_t> cacheLines;
```

- C-style memory allocation (discouraged):

```
gm_malloc, gm_calloc, gm_free, …
```

Global Memory Allocation Idioms

- Globally-allocated objects: Inherit from GlobAlloc

```
class SimObject : GlobAlloc { …
```

- STL classes that allocate heap memory: Use g_stl variants

```
g_vector<uint64_t> cacheLines;
```

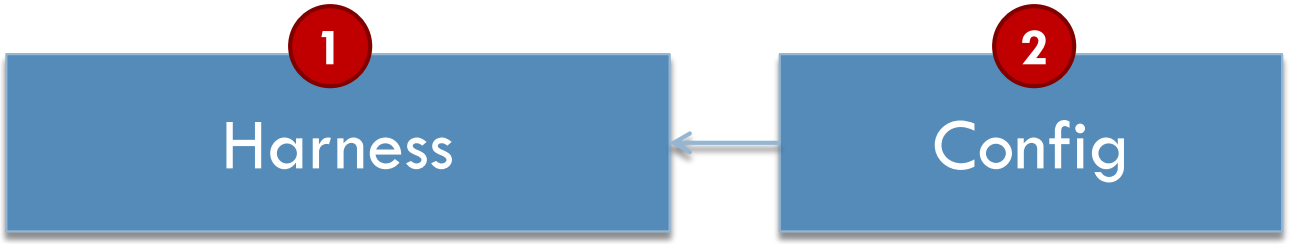
- C-style memory allocation (discouraged):

```
gm_malloc, gm_calloc, gm_free, …
```

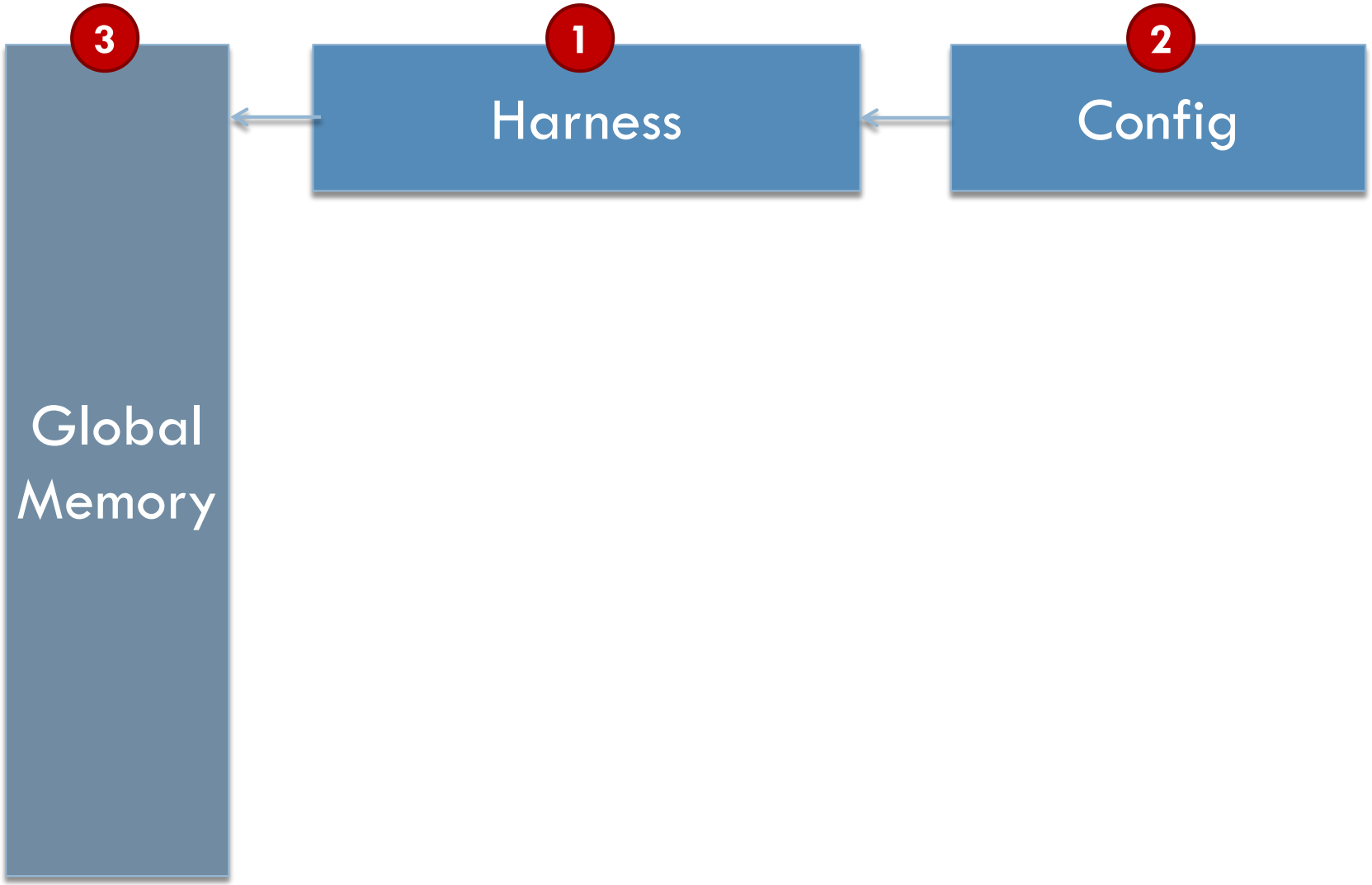
- Declare globally-scoped variables under struct zinfo

Initialization Sequence

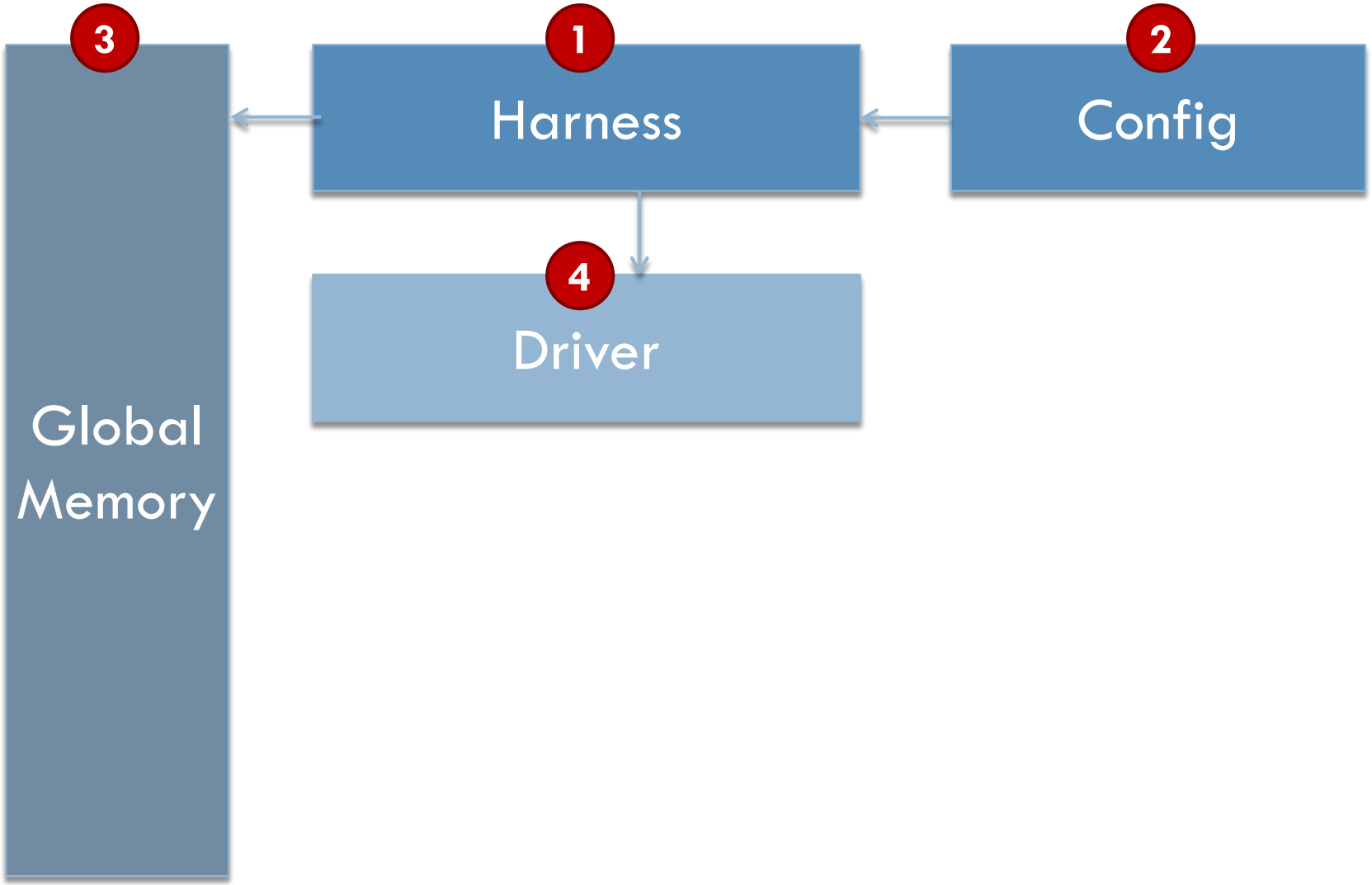
Initialization Sequence



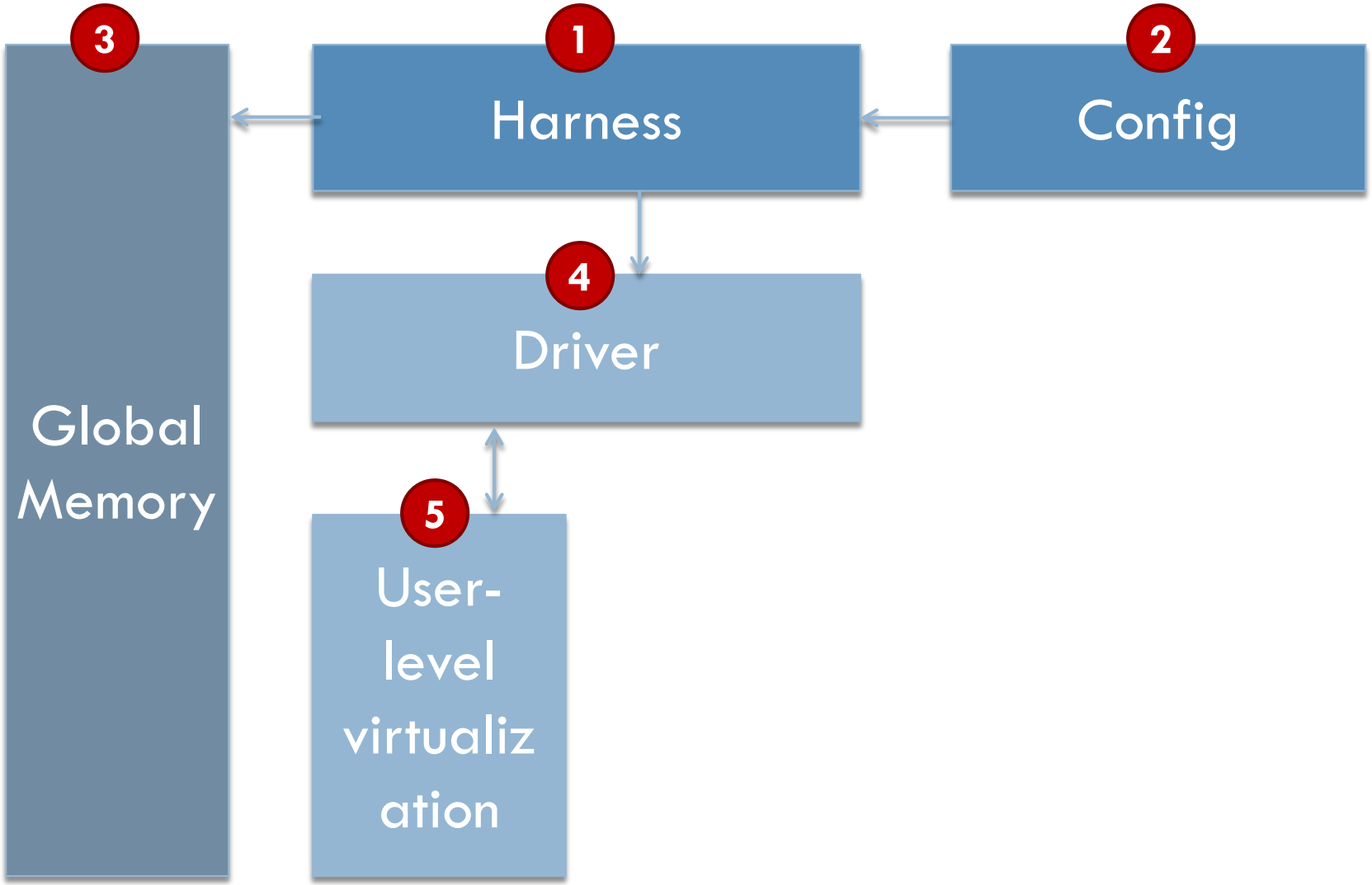
Initialization Sequence



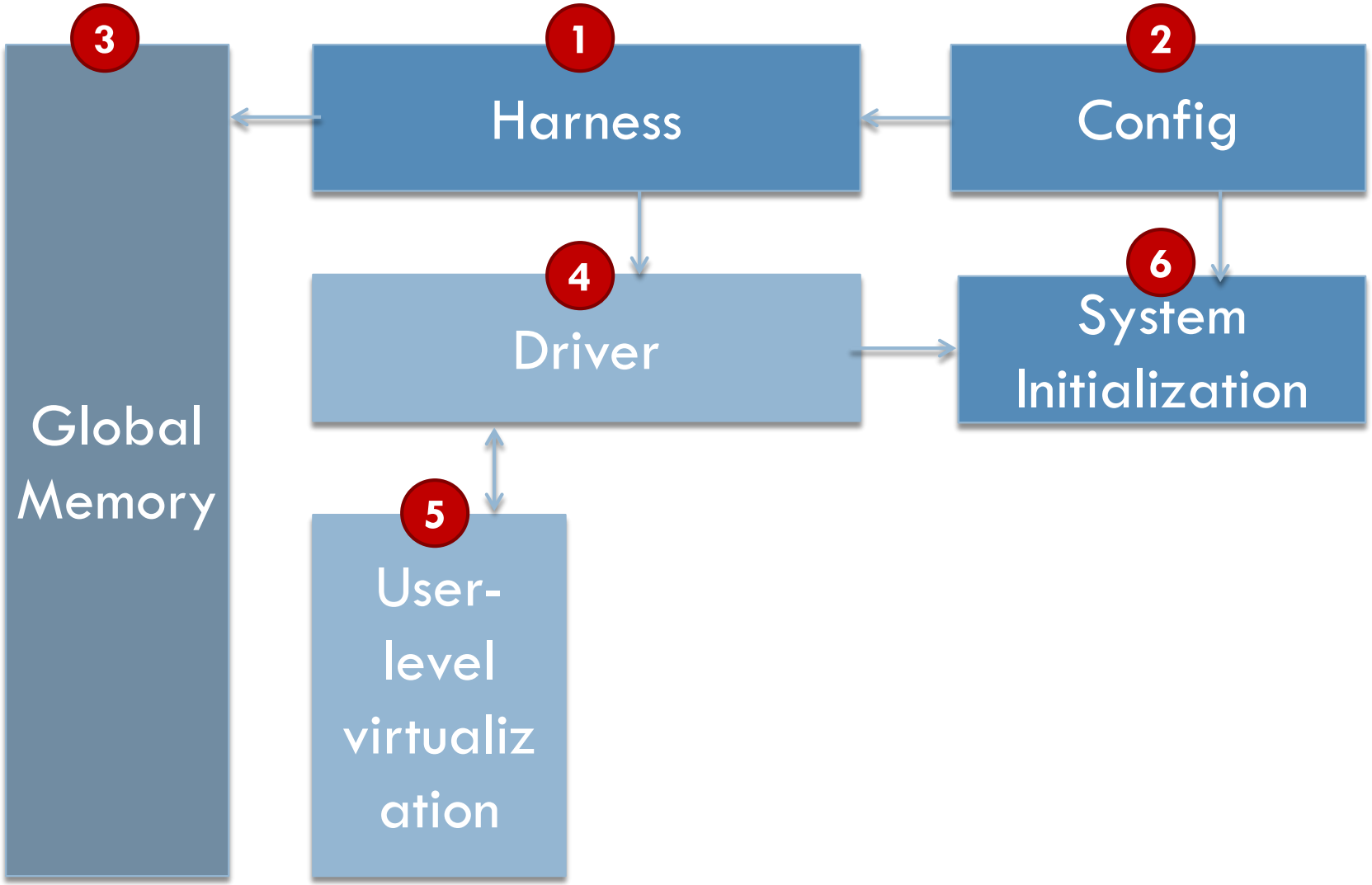
Initialization Sequence



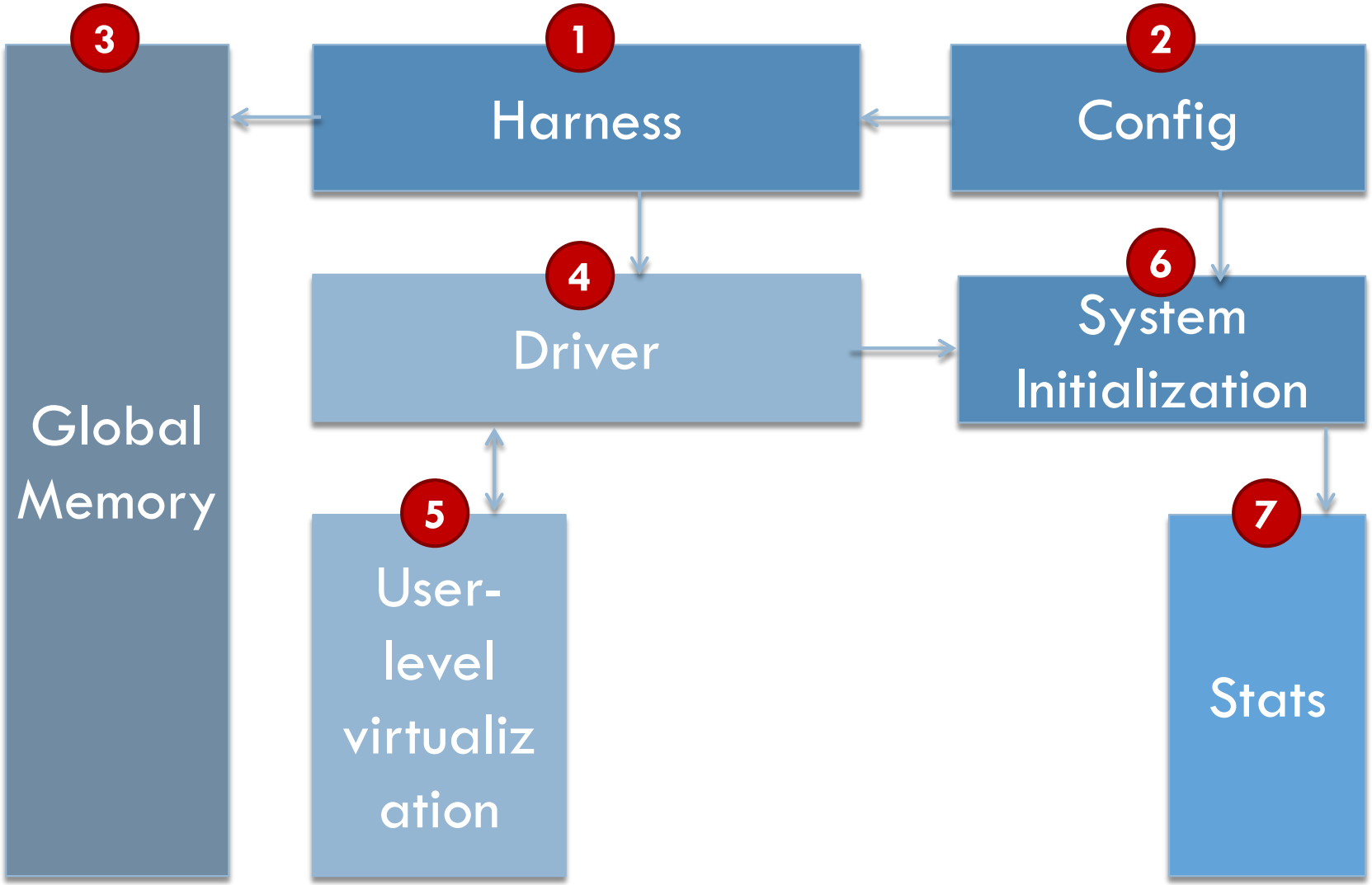
Initialization Sequence



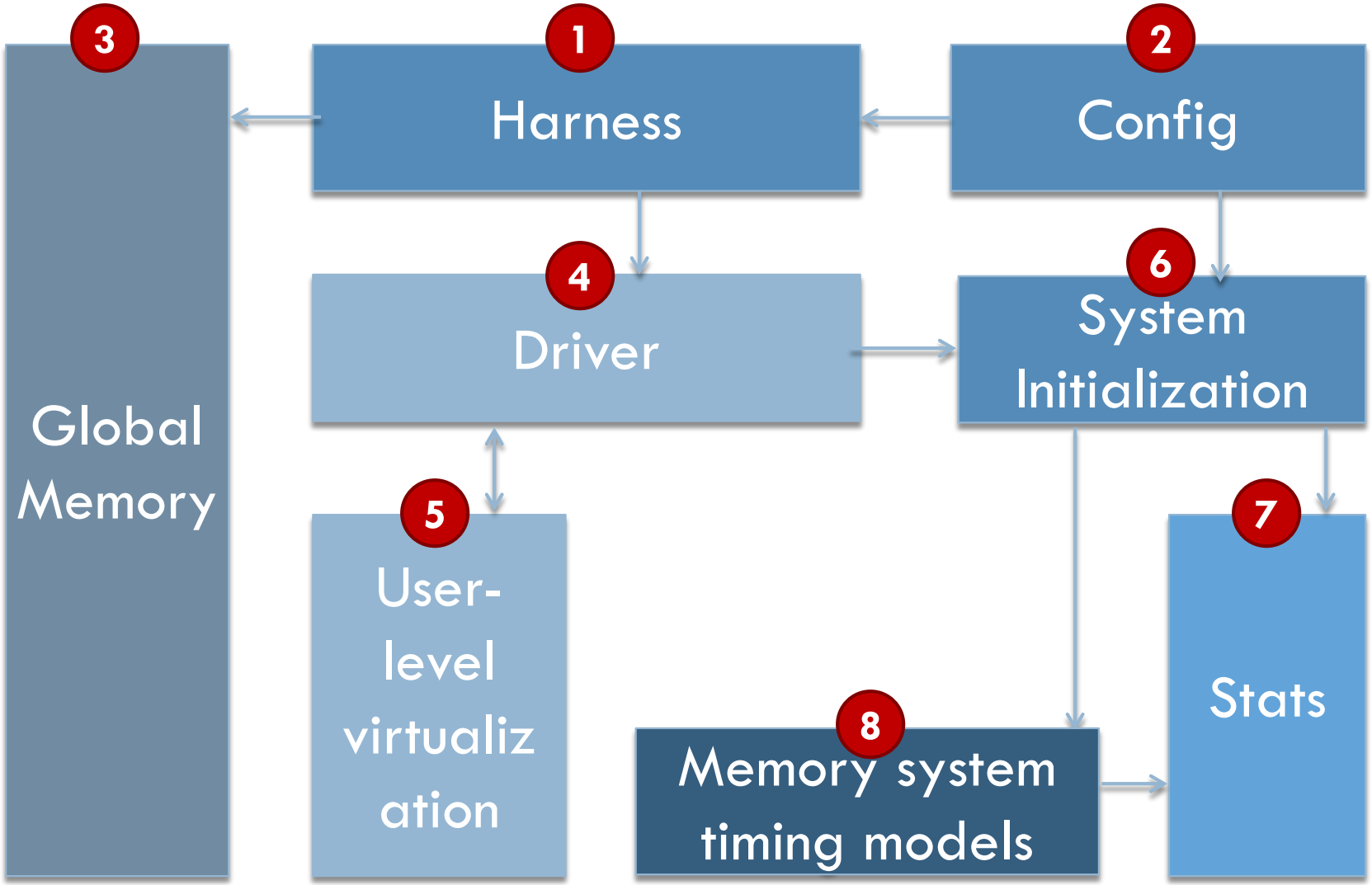
Initialization Sequence



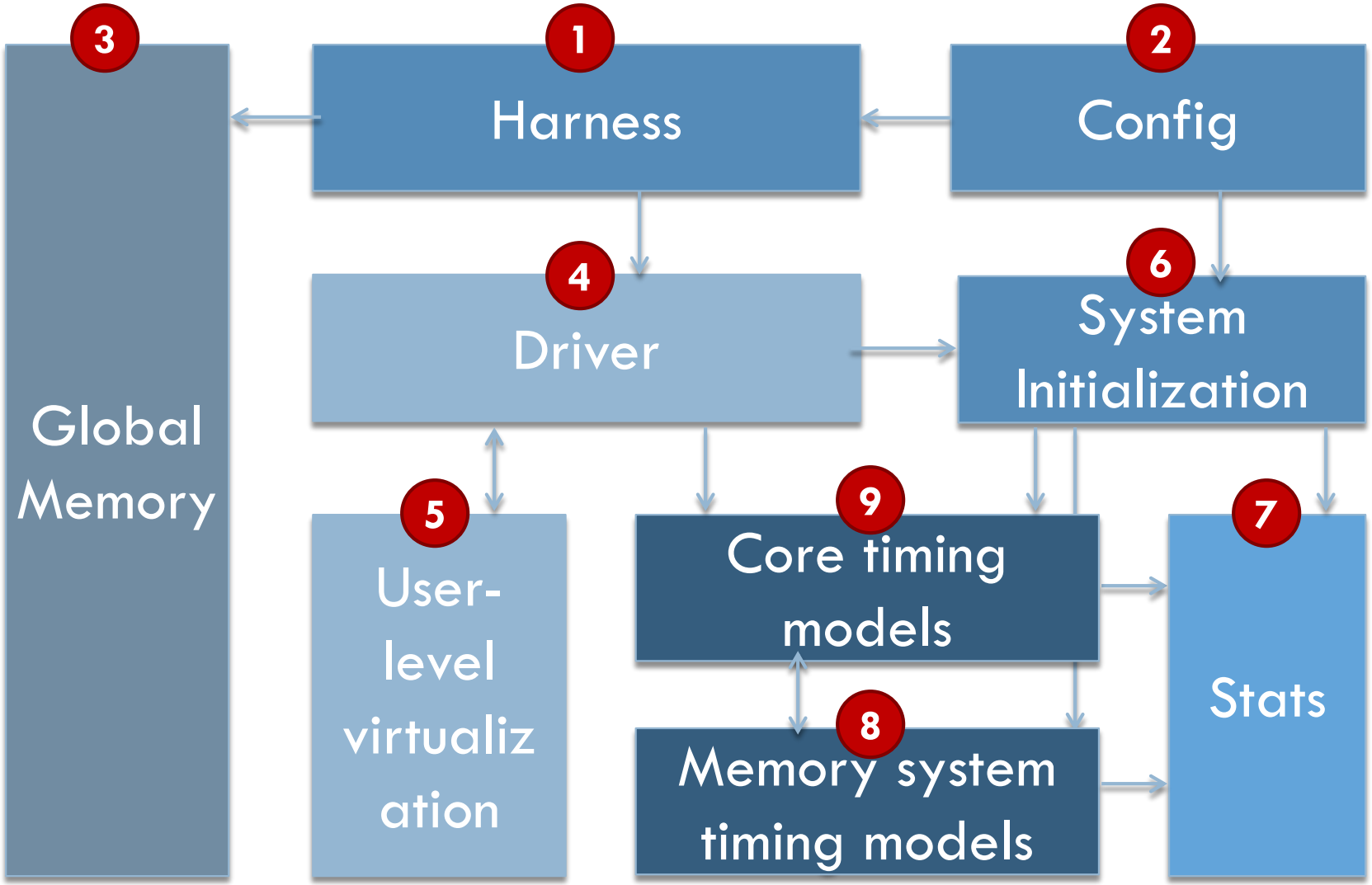
Initialization Sequence



Initialization Sequence



Initialization Sequence



Thanks For Your Attention!

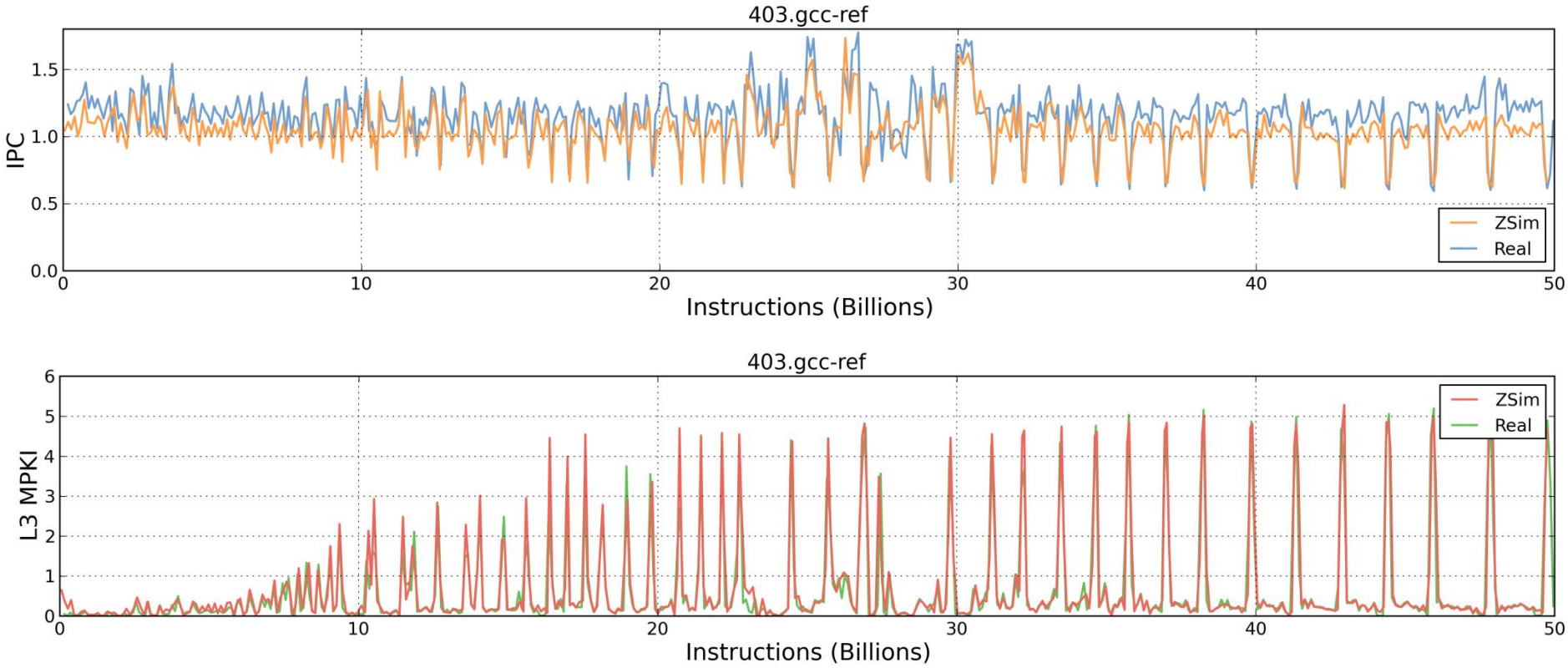
Questions?

Massachusetts Institute of Technology

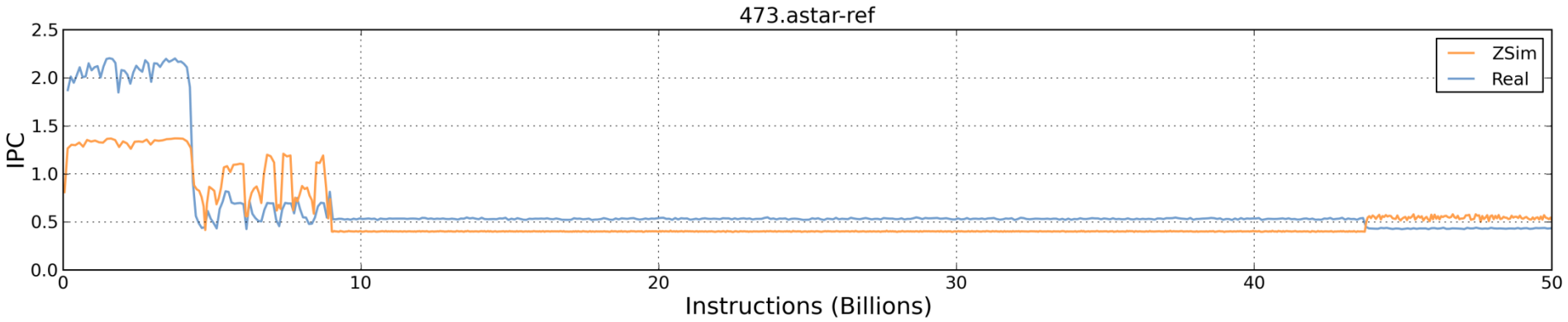
Backup Slides

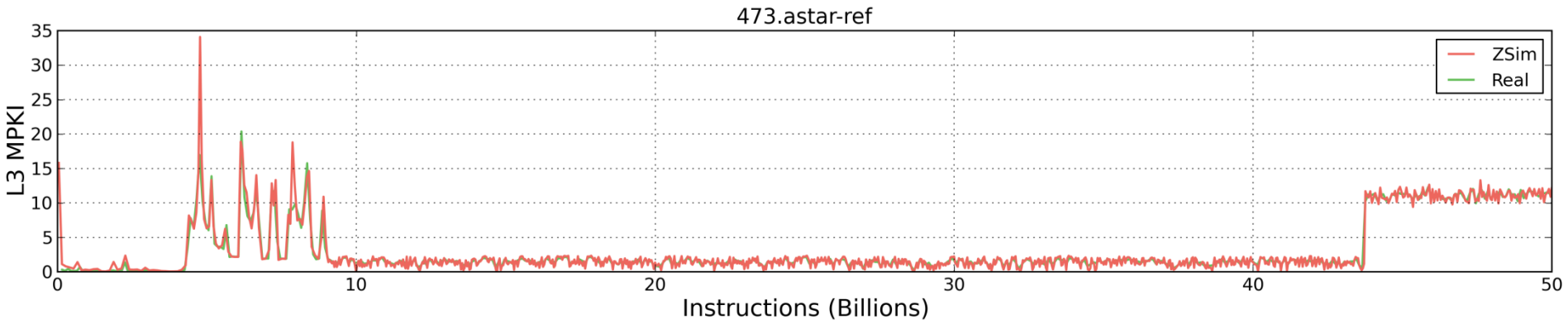
Massachusetts Institute of Technology

Single-Thread Accuracy: Traces



Single-Thread Accuracy: Traces





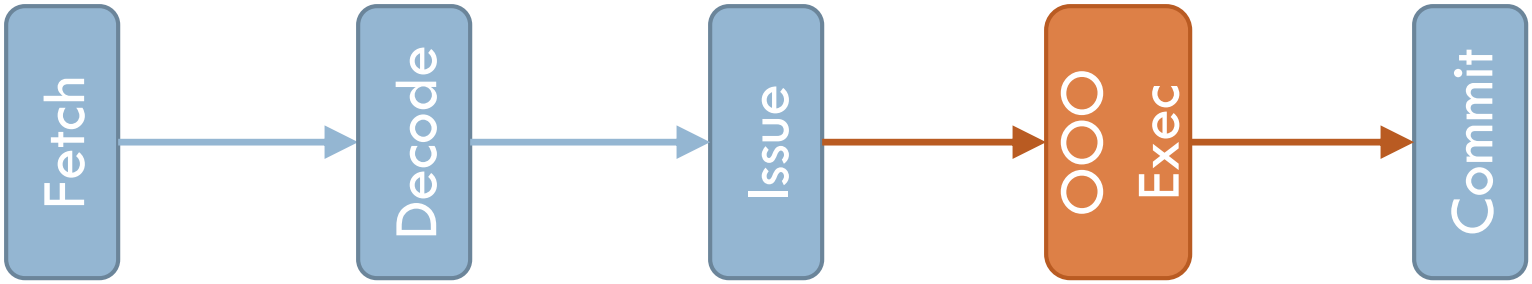
Motivation

- Timeline:
  - 2008: Decide to study 1K-core systems for my Ph.D. thesis
  - 2009: Try every sim out there, none fast enough
    - Got M5+GEMS to 512 threads [ASPLOS 2010], barely usable
  - 2010: Start developing ZSim [ZCache, MICRO 2010]
  - 2011: Make ZSim flexible, scalable, develop detailed models, other groups start using it
  - 2012: Let's publish a paper and release it…

- ZSim design approach:
  - Make judicious tradeoffs to achieve detailed 1K core sims efficiently
  - Verify that those tradeoffs result in minor inaccuracies
  - Disclaimer: Not a silver bullet & tradeoffs may not be accurate for your target system; you should validate the tradeoffs!

Instruction-Driven Timing Models

- Cycle/event-driven models: Simulate all stages cycle by cycle
- Instruction-driven models: Simulate all stages at once for each ins/uop



  - Each stage has separate clocks
  - Ordered queues (FetchQ, UopQ, LoadQ, StoreQ, ROB) model feedback loops between stages
  - Issue window tracks cycles each FU is used to determine dispatch cycle
  - Even with OOO, accurate if:
    1. IW prioritizes older uops (OK)
    2. uop exec times not affected by newer uops (OK except mem uops, ignore for now)

✓ Instr code drives directly

✓ DBT can accelerate better

✗ Harder to develop

DBT-based Acceleration

- With instruction-driven models, can push most overheads into instrumentation phase

Original code (1 basic block)

```
mov   -0x38(%rbp),%rcx
lea   -0x2040(%rbp),%rdx
add   %rax,%rbx
mov   %rdx,-0x2068(%rbp)
cmp   $0x1fff,%rax
jne   40530a
```

Basic block descriptor

Predecoder/decoder delays
Instruction to uop fission
Instruction fusion
Uop dependencies, latency, ports

Instrumented code

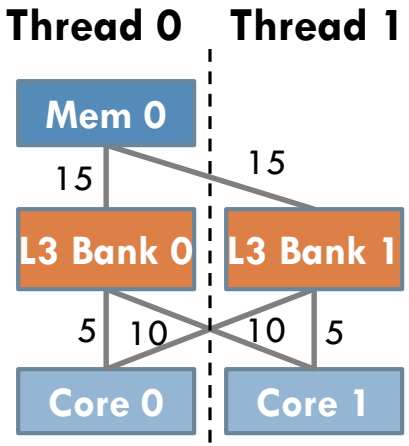
```
Load(addr = -0x38(%rbp))
mov   -0x38(%rbp),%rcx
lea   -0x2040(%rbp),%rdx
add   %rax,%rdx
mov   %rdx,-0x2068(%rbp)
Store(addr = -0x2068(%rbp))
cmp   $0x1fff,%rax
BasicBlock(DecodedBBL)
jne   10840530a
```

Type	Src1	Src2	Dst1	Dst2	Lat	PortMsk
Load	rbp		rcx			001000
Exec	rbp		rdx		3	110001
Exec	rax	rdx	rdx	rflgs	1	110001
StAddr	rbp		S0		1	000100
StData	rdx	S0				000010
Exec	rax	rip	rip	rflgs	1	000001

...

Parallelization Techniques

- Parallel Discrete Event Simulation (PDES):



Divide components across threads

Execute events from each component maintaining illusion of full order

✓ Accurate
✗ Scales poorly

Pessimistic PDES: Keep skew between threads below inter-component latency

✓ Simple
✗ Excessive sync

Optimistic PDES: Speculate & roll back on ordering violations

✓ Less sync
✗ Heavyweight

- Lax synchronization: Allow skews above inter-component latencies, tolerate ordering violations

✓ Scalable
✗ Inaccurate

Bound-Weave Parallelization

- Divide simulation in small intervals (e.g., 1000 cycles)
- Two parallel phases per interval: Bound and weave
- Bound phase:

Find paths

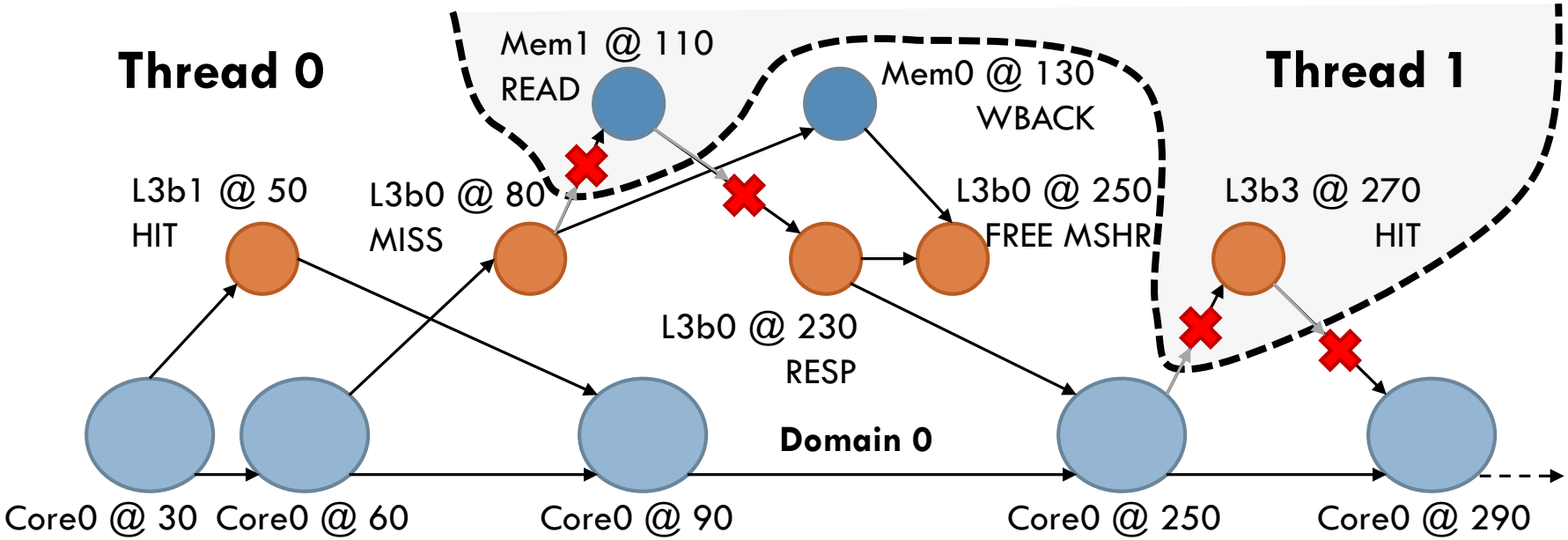
- Weave phase:

Find timings

Bound-Weave equivalent to PDES
for path-preserving interference

Bound-Weave Example

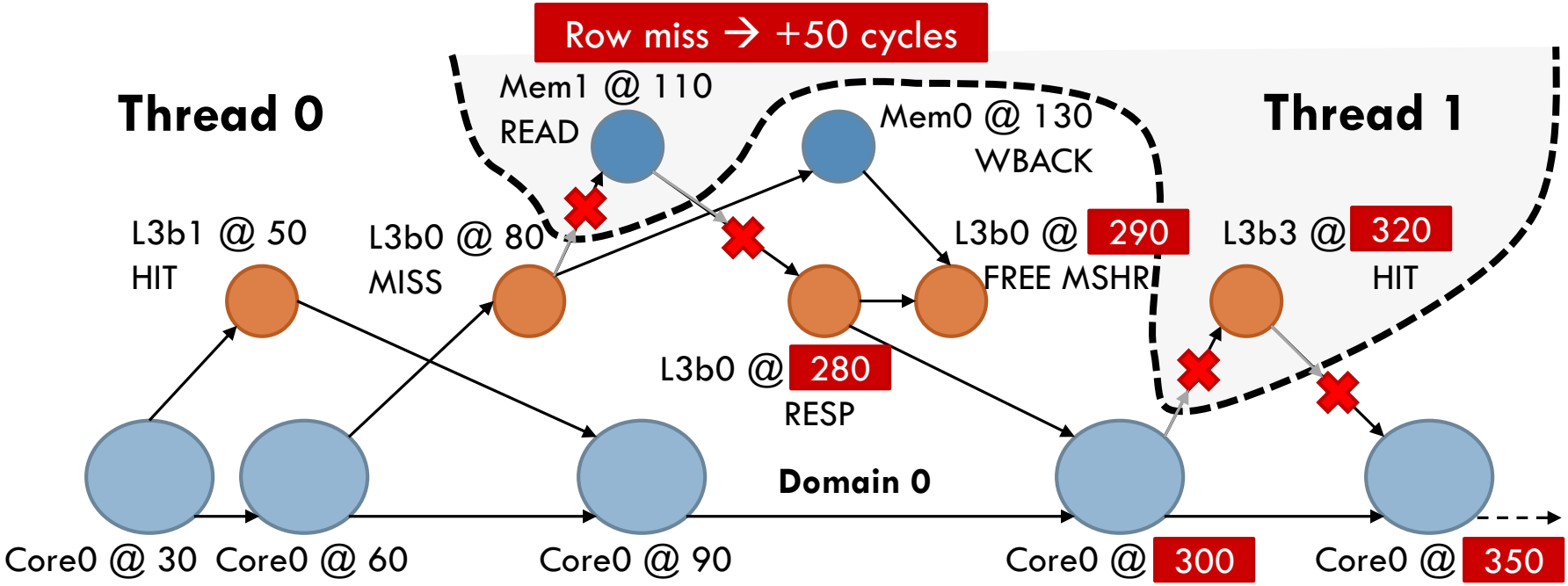
- Weave phase: Events spread across two threads



- Crossing events (✖) to only synchronize when needed
  - e.g., thread 1 reaches cycle 110, "L3b0 @ 80" not done → ✖ checks thread 0's progress, requeues itself later
  - Other synchronization-avoiding mechanisms in paper

Bound-Weave Example

- Delays propagate across crossings:



- Events are lower-bounded → No ordering violations
  - ✓ Works with standard event-driven models!
  - e.g., 110 lines of code to integrate with DRAMSim2